8086
マクロ
アセンブラ
入門
伊原充博＋小林弘幸▶著
技術評論社

8086
マクロ
アセンブラ
入門
伊原充博＋小林弘幸▶著
技術評論社

はじめに

本書は，8086アセンブラのプログラミングをするにあたり必要なハードウェアの知識や，プログラミングの考え方をわかりやすく解説しました．アセンブラについて予備知識のない人でも，抵抗なく読み進められるように多くの図と例題プログラムを用いました．アセンブラプログラミングの上達には，演習と実習を繰り返すことが重要です．本書では，最初から例題や演習問題を，身近にあるパーソナルコンピュータのWindowsまたはMS-DOS上で実行できるよう構成しました．

マイクロプロセッサが開発されて30年ほど経過し，現在は8086を源とするPentiumが広く使われています．高機能で高速なマイクロプロセッサによって，プログラムの開発技法も大きく変わり，C＋＋やJAVAなどオブジェクト指向の言語が広く使われるようになってきました．しかし，高速なプログラム実行が必要な場合や，コンピュータ周辺装置のドライバプログラムの開発には，現在でもアセンブラによるプログラミングが要求されます．

筆者らは，教育現場でコンピュータのハードウェアとソフトウェアの教育を行ってきました．その経験から，コンピュータの本質を理解するには，ハードウェアとソフトウェアの間に位置するアセンブラによってプログラムを記述することが，最も効果的であると考えるようになりました．

8086アセンブラを取り扱った本は数多く出版されてきましたが，本書は多くの図や例題，演習を通して，読者が独習できるよう構成したつもりです．本書がアセンブラを学ぼうとする読者の一助になることを願っています．

本書の執筆に当たっては，参考文献として掲げた多くの著書，文献を参照しました．これらの著者の方々に感謝すると共に，執筆の機会を与え，さらに辛抱強く原稿を校正いただいた技術評論社の皆様に感謝します．

2002年2月

第1章　8086アーキテクチャ　7

1.1　マイクロプロセッサ8086 …… 8
1.2　8086のハードウェア …… 8
1.3　レジスタ …… 10
1.4　メモリとIO構造 …… 12
1.5　データタイプ …… 15
1.6　命令セット …… 16
演習問題1 …… 20

第2章　機械語とアセンブリ言語　21

2.1　数値の取り扱い …… 22
2.2　機械語 …… 26
2.3　アセンブリ言語 …… 27
2.4　命令，疑似命令 …… 30
2.5　アドレシングモード …… 34
2.6　アセンブル，リンク，実行 …… 37
演習問題2 …… 44

第3章　データの転送命令　45

3.1　データ移動命令 …… 47
3.2　間接アドレス …… 53
3.3　アドレシング …… 59
3.4　配列のアドレシング …… 60
3.5　データの交換 …… 65
3.6　スタック命令 …… 68
3.7　テーブルによる変換 …… 74
演習問題3 …… 77

第4章　算術演算命令　79

4.1　加減算命令 …… 81
4.2　フラグと比較命令 …… 86
4.3　演算幅の拡張 …… 91
4.4　インクリメント／ディクリメント命令 …… 94
4.5　乗除算命令 …… 96
4.6　10進数の演算 …… 106
演習問題4 …… 118

第5章　ビット操作命令　119

5.1　論理演算命令 …… 120
5.2　シフト／ローテート命令 …… 124
演習問題5 …… 131

第6章　制御分岐命令　133

6.1　無条件・条件付分岐命令 …… 135
6.2　スタック …… 144
6.3　コールリターン命令 …… 146
6.4　基本サブルーチン …… 148
6.5　2進，10進，16進表示サブルーチン …… 154
演習問題6 …… 159

第7章　文字列処理　161

7.1　ストリングの考え方 …… 162
7.2　ストリング命令 …… 163
7.3　ストリング命令の使用例 …… 165
演習問題7 …… 171

第8章　入出力命令と割り込み命令　173

8.1　入出力命令 …… 174
8.2　8086の割り込み …… 176
8.3　割り込み命令 …… 178
演習問題8 …… 179

第9章　疑似命令とマクロ命令　181

9.1　疑似命令 …… 182
9.2　マクロ命令 …… 188
9.3　完全なセグメント定義によるプログラム …… 190
演習問題9 …… 194

第10章　MS-DOSシステムコール　195

10.1　システムコールの方法 …… 196
10.2　基本ファンクション …… 197
10.3　ファンクションのマクロ化 …… 200
演習問題10 …… 202

第11章 基本プログラミング 203

11.1 文字表示 …… 204
11.2 キー入力，ディスプレイ表示 …… 208
11.3 入出力の繰り返し …… 210
11.4 文字列表示 …… 211
11.5 文字列をカラー表示 …… 213
11.6 漢字の取り扱い …… 216
11.7 2進数をディスプレイに表示 …… 219
11.8 10進表示 …… 222
11.9 16進表示 …… 225
11.10 10進入力を2進コードに変換 …… 229
11.11 レジスタ表示 …… 234
11.12 メモリダンプ …… 238
11.13 アンパック型10進数の加算 …… 242
11.14 アンパック型10進数の減算 …… 248
11.15 アンパック型10進数の乗算 …… 251
11.16 アンパック型10進数の除算 …… 254
演習問題11 …… 258

付録1 8086命令一覧 …… 260
付録2 MS-DOS ファンクション一覧(抜粋) …… 265
付録3 MS-DOS コマンド一覧(抜粋) …… 266
付録4 Microsoft DDK について …… 266
付録5 DDK MASM オプション …… 268
付録6 Microsoft リアルモード Linker Version 5.60 オプション …… 269
付録7 多摩ソフトウエア社 Light Macro Assemblerについて …… 270
付録8 MASM 疑似命令一覧(抜粋) …… 270
付録9 MASM 演算子一覧(抜粋) …… 271
付録10 デバッグ …… 272
付録11 ASCII コード表 …… 276

演習問題解答 …… 277
索引 …… 284
命令および疑似命令 …… 286

第1章

8086アーキテクチャ

アセンブリ言語でプログラムを記述しようとするとき，ターゲットとなるマイクロプロセッサのアーキテクチャや，これを用いたコンピュータの構成を知ることが重要になります．特にレジスタの構成と役割，命令の構成やメモリアドレスの指定方法などをあらかじめ知っておく必要があります．アーキテクチャとは，マイクロプロセッサの構造と命令体系を含めたもののことです．

現在のコンピュータアーキテクチャには2つの大きな流れがあります．1つはCISC(Complex Instruction Set Computer：複雑命令型コンピュータ)と呼ばれる複雑な命令セットとアドレシングモードを持ったコンピュータです．その代表は8086を源とするx86アーキテクチャでしょう．もう1つはRISC(Reduced Instruction Set Computer：縮小命令型コンピュータ)と呼ばれるもので，命令とアドレシングモードを単純化してコンピュータの実行速度を高めようとするものです．その代表はマッキントッシュに用いられているPowerPCでしょう．

本章では，ターゲットである8086のアーキテクチャの理解を目的として，その概要を説明します．

1.1 マイクロプロセッサ8086

図1．1に示すように，最初のマイクロプロセッサ4004が1971年にインテル(Intel)社から世にでてから後，インテル社が8008，8080，8085を，ザイログ社がZ80を送り出し，多くの分野で8ビットのマイクロプロセッサが使用されてきました．その後1978年に16ビットのマイクロプロセッサ8086が誕生した後，さまざまなマイクロプロセッサが世に送り出されてきました．1985年には32ビットのマイクロプロセッサ80386が，1993年にはPentiumが誕生し，パーソナルコンピュータのマイクロプロセッサとして使用されています．

図1．1●マイクロプロセッサの変遷

	1971	1972	1974	1978	1985	1989	1993	2002
INTEL	4004 →	8008 →	8080 →	8086 →	80386 →	i486 →	Pentium	Pentium4
データ幅	4bit	8bit	8bit	16bit	32bit	32bit	32bit	32bit
クロック	750KHz	800KHz	2MHz	5MHz	16MHz	25MHz	60MHz	2.2GHz

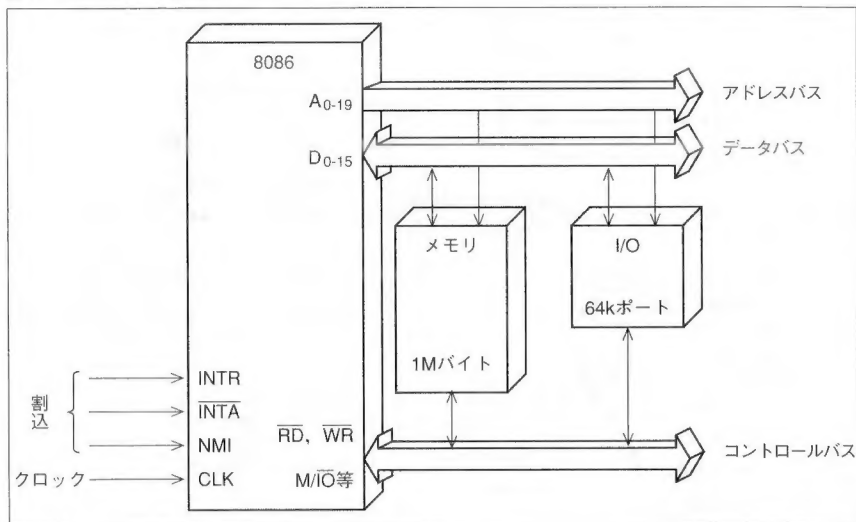
しかし，32ビットのマイクロプロセッサPentiumも実は16ビットの8086のアーキテクチャを基本として継承しており，これらすべてを含んでx86アーキテクチャと呼ぶことがあります．したがって，現在利用されているPentiumを理解するためには，その源流である8086を理解する必要があります．

1.2 8086のハードウェア

8086の特徴をまとめてみると次のようになります．

- **16ビットのデータバス**
- **20ビットのアドレスによる，1Mバイトのメモリ空間**
- **16ビットのアドレスによる，64KポートのI/O空間**
- **多数の命令**
- **リロケータブル(再配置可能)なコード**
- **多数のアドレシングモード**

8086を用いて構成したコンピュータの概要を**図1．2**に示します．8086は20ビットのアドレスバスと16ビットのデータバスを持っており，1Mバイトのメモリ空間をアクセスすることができます．I/O(input/output：入出力装置)アドレスは16ビットで，64KポートのI/Oをアクセスできます．I/O空間は一般にメモリ空間とは別に構成されます．コントロールバスには，8086とメモリ，I/O間でデータのやりとりをするための複数のコントロール信号があります．

図1．2●8086を用いたコンピュータ

8086の特徴は，実行速度を高めるため，命令をあらかじめメモリからロードしておいて待機状態にしておく，命令キューを持っていることです．命令キューは8086では6バイトですが，これをユーザが意識する必要はありません．

また8ビットの8080やZ80の命令セットに比べると，乗除算，文字列動作(ストリング)やループ動作の命令が付加されています．これらの命令によって計算処理や文字処理などが，楽に処理できるようになります．

1.3 レジスタ

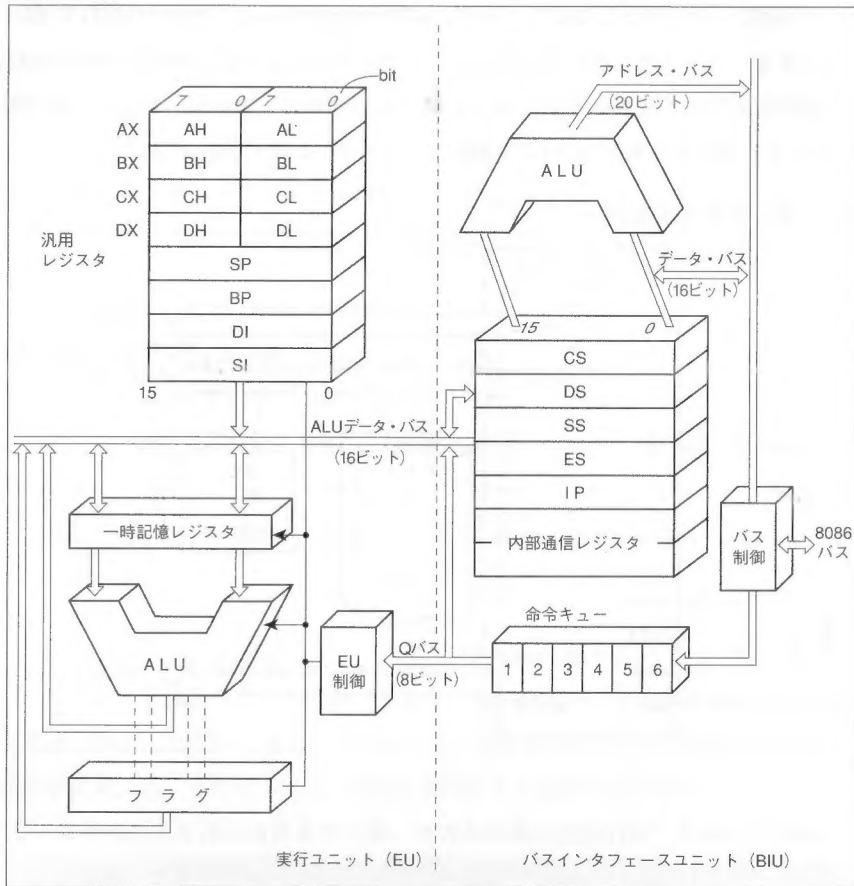
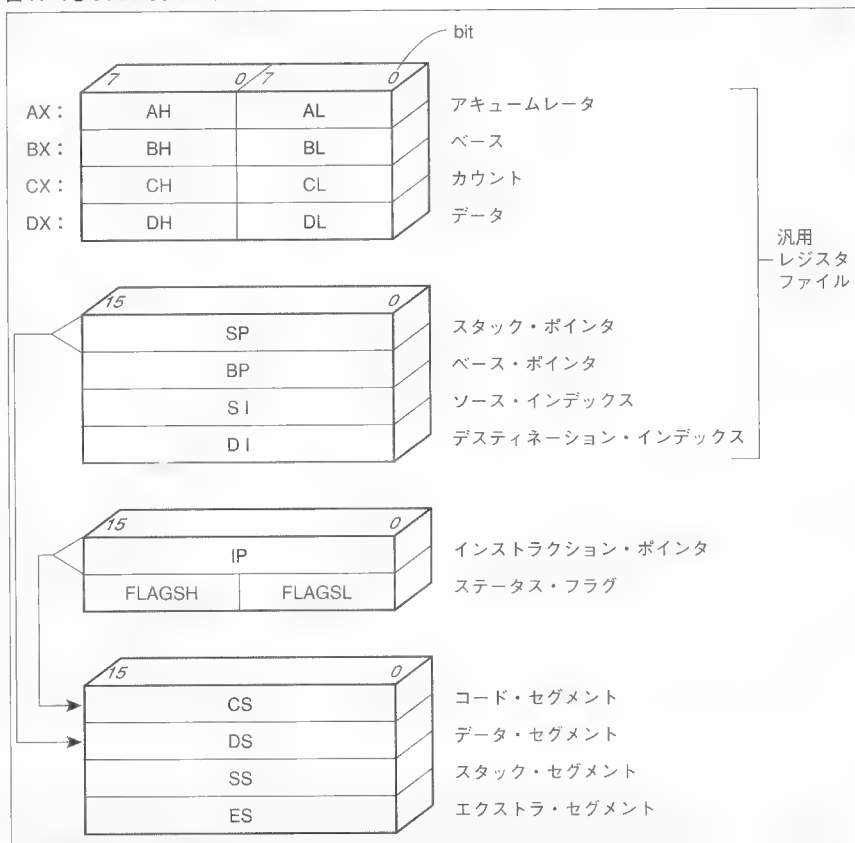
図1．3●8086の内部構成

8086は**図1．3**に示すように，16ビットのレジスタを13個とフラグ1組を持っています．汎用のレジスタには16ビットのAX，BX，CX，DXがあり，それぞれAHとAL，BHとBL，CHとCL，DHとDLの8ビットのレジスタとしても使用できます．間接アドレスに使用するものとしては，ベースポインタBPやインデックスレジスタSIとDIがあります．インストラクションポインタIPやスタックポインタSP，フラグレジスタFLAGS，メモリ管理のためのセグメントレジスタCS，DS，SS，ESなどもあります．

図1．4●8086のレジスタ

8086のレジスタAL，BL，CL，DLが，8080やZ80のA，B，C，Dレジスタに相当します．もし8080やZ80のプログラムを8086用に変更するときは，これを基準に使用レジスタを決めるとよいでしょう．

汎用レジスタの用途は，自由に決めることができますが，命令によっては用途が決まっているものがあります．カウンタが必要な命令には，自動的にCXまたはCLレジスタが使用されます．BXレジスタで間接アドレスを指定をすることもできます．

ベースポインタBPは，MOV命令でスタックを参照する場合に使用します．インデックスレジスタSIとDIレジスタは，配列や文字列を参照する際，間接アドレスを指定するレジスタとして使用します．

CS，DS，SS，ESの4個のセグメントレジスタは，後に説明するセグメントの先頭番地が，1Mバイトのメモリ空間のどこに位置するかを保持するレジスタです．1Mバイトを示すには20ビット必要ですが，セグメントレジスタは16ビットしかありません．そこで実際のアドレスが生成されるときは下位に4ビットの0が加えられて，20ビットに拡張されます．

通常プログラムカウンタは，次に実行される命令のあるメモリ番地を保持します．8086ではオフセット(セグメントの先頭からのアドレス差)を持つためインストラクションポインタIPと呼ばれます．

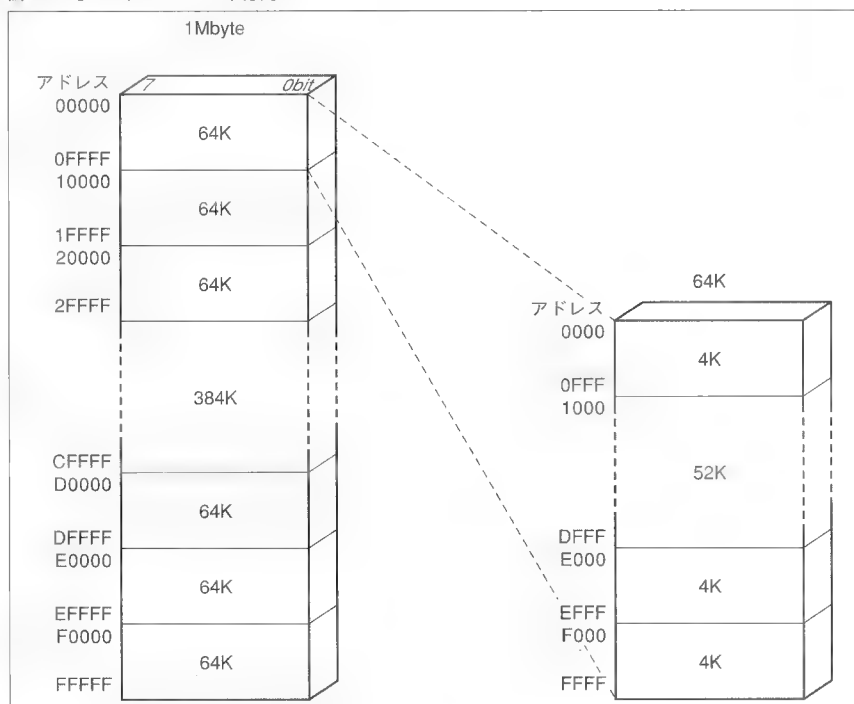
8080のフラグレジスタを継承した，AF(補助キャリ)，CF(キャリ)，PF(パリティ)，SF(サイン)，ZF(ゼロ)フラグの他に，DF(方向)，IF(割り込み許可)，OF(オーバフロー)，TF(トラップ)フラグがあります．

以上述べたように，8086は8ビットのマイクロプロセッサ8080と8085のアーキテクチャを色濃く残しています．

これらすべてのレジスタの役割を知ることは大変ですが，実際に8086のプログラムをアセンブラで記述すると，これでもレジスタの数が不足することがあり，もう少し汎用レジスタの数が多ければプログラムを楽に記述できるのに…，といった場合が出てきます．

1.4 メモリとIO構造

8086は20ビットのアドレスを持っており，**図1．5**に示すように1Mバイトのメモリ空間をアクセスすることができます．

図1．5●メモリのセグメント分割

8086のメモリは，論理的にコードセグメント，データセグメント，スタックセグメント，エクストラセグメントの4つの領域に分かれます．各セグメントには，次のようなものが格納されます．

- **コードセグメント　　：コード(機械語命令)**
- **データセグメント　　：データ**
- **スタックセグメント　：スタックデータ**
- **エクストラセグメント：補助データ**

これは，コードやデータなど，性質の異なるものの領域を明確に分けたり，1Mバイトのメモリを16ビットのレジスタ群でアクセスできるようにしたりするためと思われますが，8086のプログラミングをわかりにくくしている原因でもあります．

図1．6●8086のメモリ空間

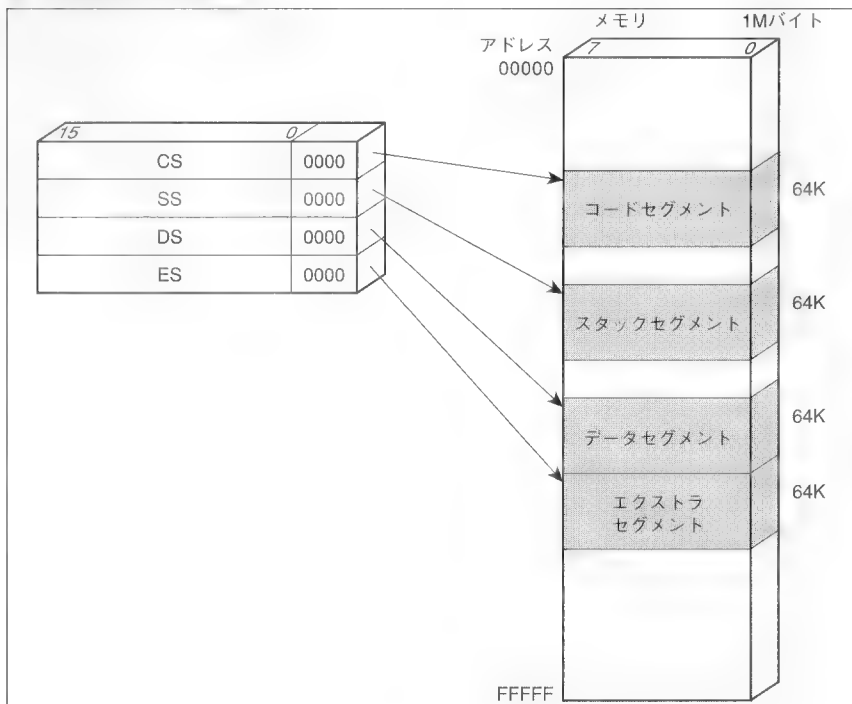



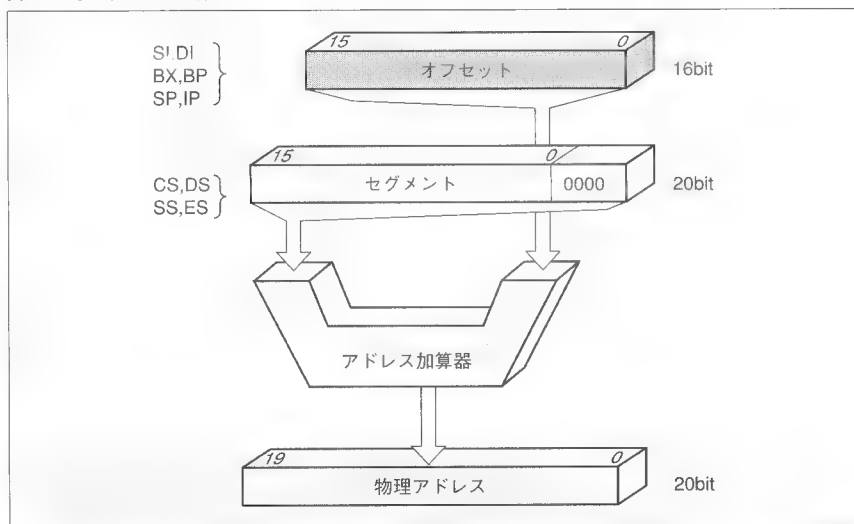
図1．6に示すように，それぞれのセグメントは64Kバイトで構成され，下位4ビットが0000_2のメモリアドレスならば，実メモリのどこにでも配置することができます．各論理セグメントを，実メモリに割り当てるのが4つのセグメントレジスタで，各セグメントの先頭アドレスを保持します．

物理的な20ビットのアドレスを計算する機構を**図1．7**に示します．セグメントの先頭からの距離であるオフセットとセグメントレジスタを4ビットシフトした値とが加算され，物理アドレスが求まります．

データバスが16ビットの8086では，メモリ上のワード(2バイト)データをアクセスするときは，そのデータアドレスが0，2，4・・・のように偶数番地から始まっている場合の方が，1，3，5・・・のように奇数番地から始まっている場合に比べ，より速くアクセスできることがあります．これは8086が，常に偶数番地と次の奇数番地の2バイトをデータとしてアクセスすることから生じます．奇数番地から始まるワードデータは2回に分けてアクセスされることになる

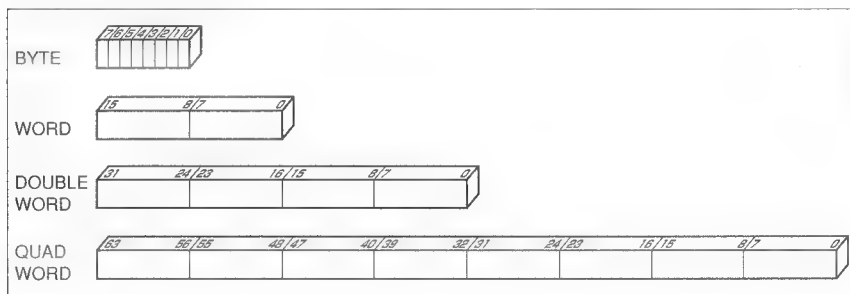
ためです.

一般に8086用に作成したプログラムをROMに書き込むときは，プログラムを偶数番地と奇数番地に分けて別のROMに書き込みます．さらにワードデータは偶数番地から配置されるように注意しないと，効率よく処理することができません．同様に，8086ではメモリと同様に奇数番地から始まるI/Oポートのワードデータは，2回に分けてアクセスされてしまいます．このためデータバスが16ビットであることを生かすには，I/Oの場合もアクセスに注意する必要があります.

図1．7●アドレスの生成

1.5 データタイプ

8086のデータは，基本的に8ビットか16ビットの整数データです．8ビットのデータをバイト(byte)，16ビットデータをワード(word)と呼んでいます．ただし，一般にワードという語が16ビットとして認知されているわけではなく，x86アーキテクチャでの独特の呼び方です．さらにワードデータの乗算などの結果，32ビットのデータが生ずることがありますが，これをダブルワード(double word)と呼ぶことがあります.

文字データを扱う場合，多くの場合1バイトの整数データとして扱うこととなります．

図1．8●データタイプ

8086の基本演算としては整数のみが用意されており，実数計算はコプロセッサと呼ばれる計算専用のICが担当します．もし実数を扱う場合は，ユーザが実数データを構成する必要があります．現在のPentiumの場合，マイクロプロセッサ自身が実数計算します．

8086にはストリングと呼ばれる文字列データがあります．メモリ上の一連のバイトまたはワードの文字列などの処理を強力に行うストリング命令が用意されています．

1.6 命令セット

8086の命令セットを分類すると次のようになります．

(1)データ転送命令

(2)算術演算命令

(3)ビット操作命令

(4)ストリング命令

(5)制御分岐命令

(6)プロセッサ制御命令

このうち，8086として特徴的なのは，文字処理を目的としたストリング命令でしょう．また8086では，データを8ビット(バイト)でも16ビット(ワード)で

も処理できます．

データ転送命令は多くのアドレシングモードによって，変数，配列，構造体，ローカル変数など簡単な構造のデータから複雑な構造のデータまで，アクセス可能になっています．

算術演算命令では，整数の加減乗除算命令が用意されています．実数の演算命令はなく，ソフト的に対応するか，コプロセッサを利用することになります．8086では8または16ビットデータに対し，有符号か無符号かで計算できるIMUL，MUL，IDIV，DIV命令が用意されています．

ストリング(文字列)命令は，ストリングと呼ばれるメモリ上の一連のバイトまたはワードデータや文字列などの処理を強力に行います．たとえば，MOVS命令はストリングを移動する命令で，設定したカウンタの増減やソースアドレスとディスティネーションアドレスの増減を自動的に行います．さらにプリフィックスと呼ばれる特殊な1バイト命令を付け加えると，カウンタがゼロになるまで命令の実行を繰り返します．

分岐命令は絶対番地ではなく，インストラクションポインタ(IP)からの相対番地を指定します．これによりプログラムをメモリのどの位置に置いても動作する再配置可能(relocatable)なプログラムを記述することができます．

条件分岐命令は，多くの条件で分岐が可能になっています．これはデータを無符号データと有符号データとに明確に分けて取り扱うためです．たとえば，1バイトのデータ「1111 1111」は無符号ならば「255」ですが，有符号ならば「－1」です．無符号データを比較してジャンプする命令がJA(jump above)やJB(jump below)です．有符号データを比較してジャンプする命令がJG(jump greater)やJL(jump less)です．符号のない絶対アドレスなどを比較するとき，有符号を比較するジャンプ命令を用いると思わぬ結果になるので注意が必要です．

これらの命令は，1～6バイトで構成されることになります．命令によって複雑に命令長が変化します．同じような命令でも，メモリアクセスかレジスタアクセスかなどによって変わります．**図1．9**に8086の命令形式を示します．どの命令も最初の1バイトは命令自身を示すコードになっています．残りは，データやデータアドレスを示すことになります．

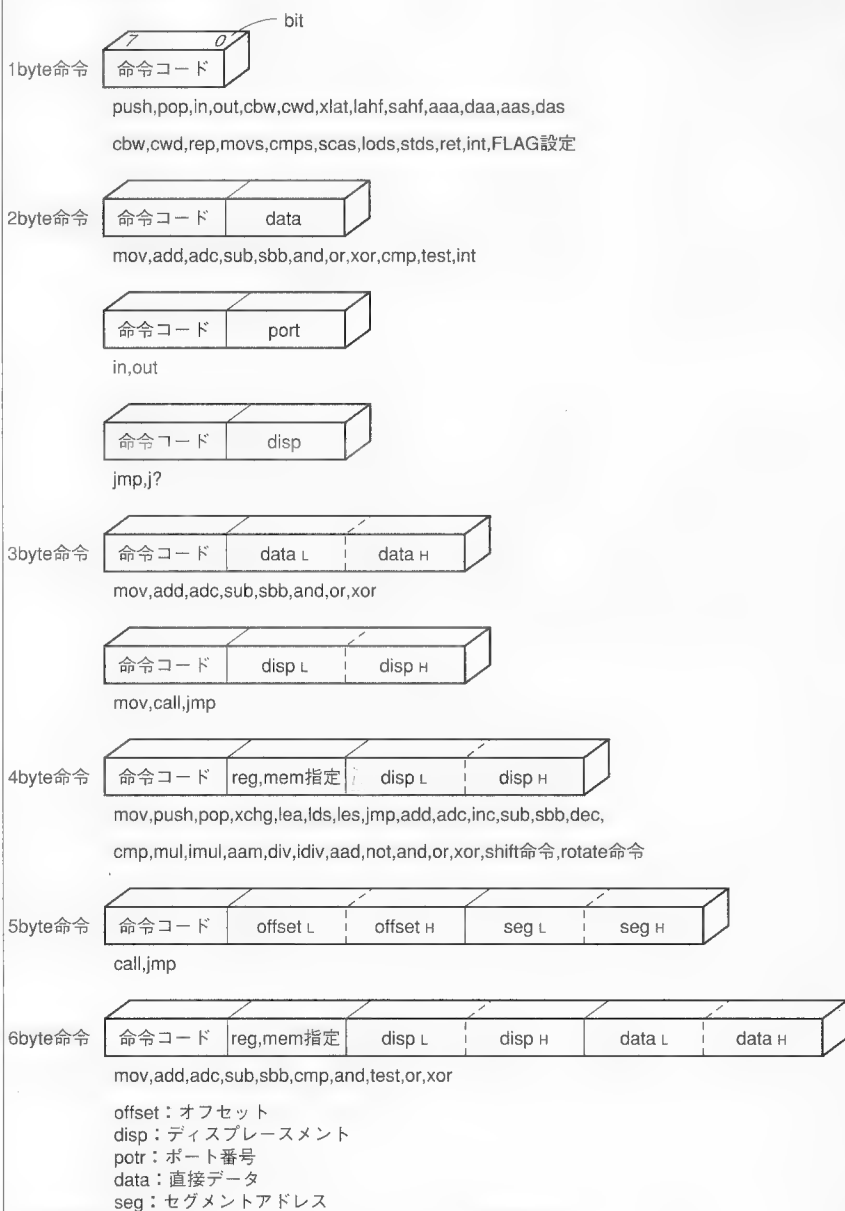
1バイト命令を見ると，命令コードだけでできています．実際の命令としては，push，pop，in，out，cbw，cwd，int命令などがあります．

2バイト命令の場合，第1バイトは命令コード，第2バイトは8ビットのデータ，メモリアドレス，I/Oポート番号などになります．2バイト命令には，in，out，cmp，条件付jump命令などがあります．1バイト命令と同じくin，out命令があります．これらは入出力ポートをどのように指定するかによって異なってきます．3～6バイト命令の場合も同じ命令が出てきますが，このような理由から異なったバイト数となります．

3バイト命令の場合は，第2，第3バイトは，2バイトのデータやアドレスを示すことになります．この命令には，mov，add，call，jump命令などがあります．

4バイト命令は，3バイト命令と同じく2バイトのデータやアドレスを，第3，第4バイトに持っています．第2バイトはレジスタやメモリアクセスを示すコードとなっています．

6バイト命令は，第5，6バイトが2バイトのデータそのものを持っています．また第3，4バイトはメモリアドレスを保持しています．第1バイトは命令コードで，第2バイトはアクセス先がレジスタかメモリかなどを示すバイトとなっています．

図1．9●8086の命令形式

演習問題 1

1. 8086について，次の問いに答えよ．
   (1) 汎用レジスタをあげ，それぞれの役割をのべよ．
   (2) 一般的なコンピュータのプログラムカウンタの役割をする8086のレジスタは何か．
   (3) インデックスレジスタを示し，その役割を示せ．
   (4) スタックはどのように構成されるか．スタックトップを指すレジスタは何か．
   (5) データセグメントを80000H番地を先頭にしたいとき，設定すべきレジスタとその値は何か．
   (6) スタックセグメント，エクストラセグメント，コードセグメントの先頭を指すレジスタを示せ．
   (7) byte，word，double wordのビット数を示せ．
   (8) 機械語は何バイトで構成されるか示せ．
   (9) メモリ空間は何バイトか示せ．
   (10) I/O空間は何ポートか示せ．
   (11) セグメントとその必要性を示せ．

第 2 章

機械語とアセンブリ言語

本章ではアセンブリ言語に必要な数値の取り扱い，8086 機械語の構成，プログラム記述法とアセンブル，リンクと実行方法について説明します．

読者がプログラムを記述し，これを実際に走らせてみることが，8086 アセンブリ言語を習得する近道です．実行形式になったプログラムは，MS-DOS 上で実行することになりますが，この実行方法についてもこの章で詳しく説明します．

2.1 数値の取り扱い

アセンブリ言語ではプログラムや機械語の表現に，2，8，10，16進数等多くの数値を取り扱います．ここではこれら表記のしかたを簡単に説明しましょう．

通常用いられている10進数(decimal number)で，165という数値を例として考えてみましょう．165の各桁は，下に示すように各桁に対応した10^2=100，10^1=10，10^0=1の重み(weight)を持っています．

$$165_{10}=1\times10^2+6\times10^1+5\times10^0$$

一方ここで，165の添字10は，10進数を意味しています．重みの基本となる10を基数(radix)と呼びます．また10進数で用いられる数字は，0から9までの10種類となります．

コンピュータ内部では，2進数(binary number)が用いられています．2進数も，10進数と同じように重みを持った各桁で表現されます．2進数では基数が2となり，用いられる数字は0と1の2種類となっています．10進数の165は，2進数では次式で表現されます．なお添字2は，2進数であることを意味しています．

$$10100101_2=1\times2^7+0\times2^6+1\times2^5+0\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0$$
$$=165_{10}$$

2進数の各桁はbit(binary digitの略)と呼ばれ，2のべき乗の重みを持っています．アセンブリ言語では2進数をよく用いるので，2^nの重みと呼称法を**表2.1**に示します．この呼称法は，記憶しておくと便利なことが多いのでここに示しましたが，自分の呼びやすい方法をとればよいでしょう．

なおコンピュータ内部でのアドレスやデータの表現では，1Kは1000ではなく，1024として取り扱うのが普通です．また2進数が何桁で構成されていても，その最上位桁をMSB(most significant bit)，最下位桁をLSB(least significant bit)と呼びます．

表2．1●2進数の重み

2^n	10進数	呼　称
2^0	1	イチ
2^1	2	ニ
2^2	4	ヨン
2^3	8	パ
2^4	16	イチロク
2^5	32	ザンニ
2^6	64	ロクヨン
2^7	128	イチニッパ
2^8	256	ニゴロ
2^9	512	ゴイチニ
2^{10}	1,024	1K
2^{11}	2,048	2K
2^{12}	4,096	4K
2^{13}	8,192	8K
2^{14}	16,384	16K
2^{15}	32,768	32K
2^{16}	65,536	64K
2^{17}	131,072	128K
2^{18}	262,144	256K
2^{19}	524,288	512K
2^{20}	1,048,576	1M

2進数を用いる場合，桁数が多くなると実際にはわかりにくくなります．そのため，2進数を16進数(hexa decimal number)に変換し，桁数を少なくしてわかりやすくすることがよくあります．16進数では，0～9の数字とA～Fの文字が16進数字として用いられます．10～15までの数を1字で表現するために，A～Fまでの文字を数字の変わりに使用しているのです．

10進数の165は，16進数ではA5となり，次式のように表現されます．

$$\mathbf{A5_{16}=10\times16^1+5\times16^0=165_{10}}$$

4ビットで表現できる2進数の範囲で，10進数，2進数，16進数の対応表を**表2．2**に示します．16進数は2進数を4桁ごとに区切って，これを16進数の1桁として0～9とA～Fのいずれかで表現します．

表2．2●10進数，2進数，16進数の対応

10進数	2進数	16進数
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

表2．3●2進数，16進数への変換

10進数	165_{10}
2進数	1010 0101
16進数	A 5

コンピュータ内部では，整数は符号のないデータとして取り扱う場合と，符号のあるデータとして取り扱うことがあります．10進数で負の数を表現するには，「－165」のように絶対値に－の符号を付けて表現します．しかし，コンピュータ内部では2進数が用いられており，＋や－の符号を用いることはできません．いずれかを符号ビットとして，0か1によって＋と－を表現する必要があります．通常コンピュータ内部では，負の数を表現するために，2の補数(2's complement)が用いられています．この場合，2進数の最上位桁(MSB)の0または1は正負の符号を示すことになります．負数を2の補数で表現した2進数では，MSBが0ならば正の数，1ならば負の数となります．

2の補数は次の手順に従って求めることができます．

①補数表現しようとする数を，正の2進数で表現

②2進数の各桁の1を0に，0を1に反転

③LSBに1を加算

なお，②で求めた結果は1の補数と呼ばれます.

10進数の「−67」を2の補数表現する例を，**図2．1**に示します.

図2．1●2の補数の求め方

	−67	2の補数表現する数
①	01000011	正の2進数で表現
②	10111100	0/1を反転→1の補数
③	10111101	LSBに1を加算→2の補数

1バイト(8ビット)のデータで表現できる整数の例を，**表2．4**に示します．1バイトのデータは，符号のないデータ(すべてが正)とみるか，符号のあるデータ(正負)とみるかで，表現する数が異なってきます．網掛けをしたMSBが0である場合は，0〜127までとなって，符号付きと符号なしデータではまったく同じ値となります.

MSBが1となった場合は表現する値が異なります．10000000は，符号なしでは128を，符号付きでは−128を表現しています．11111111は，符号なしでは255を，符号付きでは−1を表現していることになります.

符号付きデータの場合，正の数は最大127(01111111)までで，負は最大−128(10000000)までとなって，負で表現できる数は正で表現できる数に比べ1つ多くなります.

表2．4●整数表現法(1バイト)

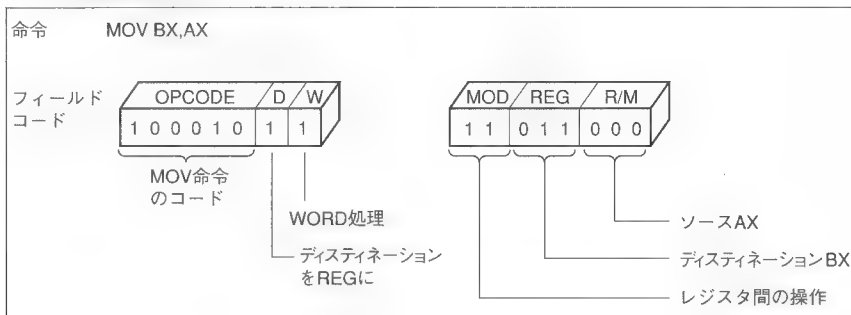
MSB 2進数 LSB	符号なし	符号付
0 0 0 0 0 0 0 0	0	0
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
⋮	⋮	⋮
0 1 1 1 1 1 1 1	127	127
1 0 0 0 0 0 0 0	128	−128
1 0 0 0 0 0 0 1	129	−127
⋮	⋮	⋮
1 1 1 1 1 1 1 0	254	−2
1 1 1 1 1 1 1 1	255	−1

2.2 機械語

コンピュータが直接理解できる言葉のことを，機械語と言います．言ってみれば，これは1と0の数字のつながりです．実際の回路上では，この1と0が電圧のHighとLowに置き換えられます．

8086の機械語命令は命令ごとにその長さが異なっています．8086の命令長は1〜6バイトで構成されており，同じデータ移動命令でもレジスターレジスタ間か，レジスターメモリのデータ転送かで，命令長が異なります．

例として，AXレジスタの内容をBXレジスタへ移動する命令(mov bx,ax)をみてみましょう．**図2．2**に示すように，機械語は第1バイトと第2バイトの2バイト構成になっています．さらに第1バイトは3つのフィールドからなっています．movという命令を示すフィールドOPCODE，移動先が命令の第2バイトのREGフィールドに示されているフィールドD，データがバイトかワードかを示すフィールドWです．第2バイトは，レジスタ間の操作を示すフィールドMOD，ディスティネーションレジスタがBXであることを示すフィールドREG，ソースレジスタがAXであることを示すフィールドR/Mです．これらのフィールドはソースかディスティネーションかによって，またはデータがバイトかワードかによって複雑に変化します．もし人がプログラムを直接機械語で書くとしたら，機械語を構築するという大変な作業が必要となってしまいます．

図2．2●機械語命令

8ビットのマイクロプロセッサ8080やZ80の時代には，アセンブリ言語のソースプログラムを人の手で機械語に変換するハンドアセンブルが可能でした．ニーモニックを聞くと直ちにこの機械語を返してくる，「人間アセンブラ」と呼ばれるような人も実際に存在しました．

しかし，複雑になった8086の機械語構成では，ハンドアセンブルすることはかなり困難です．アセンブリ言語のソースプログラムを実行するマイクロプロセッサは存在しません．アセンブリ言語のソースプログラムは，必ず8086の機械語に変換する必要があります．このソースプログラムを機械語に変換してくれるのがアセンブラです．

2.3 アセンブリ言語

アセンブリ言語とは，機械語で命令を記述していく操作を簡単にするために生まれたもので，機械語の1010という数字の連なりを，人にわかりやすいシンボルに置き換えて記述します．したがってアセンブリ言語の命令は，1対1で機械語命令に置き換えることができます．実際にコンピュータを動かす場合は，アセンブリ言語のソースプログラムを機械語のオブジェクトプログラムに変換する必要があり，このアセンブリ言語プログラムを機械語に変換してくれるのが，アセンブラと呼ばれるアセンブリ言語処理系になります．

アセンブリ言語は，機械語命令をこれに関連したわかりやすいシンボルで表します．たとえば加算はadd(add)，減算はsub(subtract)，データの移動はmov(move)などです．多くのシンボルはその処理を示す単語の一部をとった形になっています．

アセンブリ言語の記述例を**図2．3**に示します．アセンブリ言語では，命令とそれに必要なデータを保持しているレジスタやメモリアドレスを指し示す必要があります．基本的には1行に1命令を書きます．1行に2つ以上の命令を書くことはできません．

機械語命令をわかりやすいシンボルに置き換えたものを，ニーモニックと呼びます．レジスタやメモリ番地を示すシンボルのことを，オペランドと呼びます．データをある場所から別の場所に移す場合，送り元と送り先(目的地)が必

要になります．送り元をソース(source)，送り先をディスティネーション(destination)と呼びます．図2．3の例では，第1オペランドのBXがディスティネーションで，第2オペランドのAXがソースとなっています．

記述が，ディスティネーション,ソースの順になっていることに注意してください．これは，8086マイクロプロセッサの開発元であるインテル社のマイクロプロセッサ共通の記述法です．

最初にある「label0:」のように:記号を付けられたラベルは，この命令があるメモリ番地を示すために用いられます．たとえばこの命令までジャンプするような命令を書く場合，プログラマがメモリ番地を計算する必要はなく，「jmp label0」と書けばよいことになります．すべての命令にラベルを付ける必要はなく，番地を示す必要がない場合は，ラベルを記入する必要はありません．ほとんどの命令で，ラベルの部分は何も記入されません．

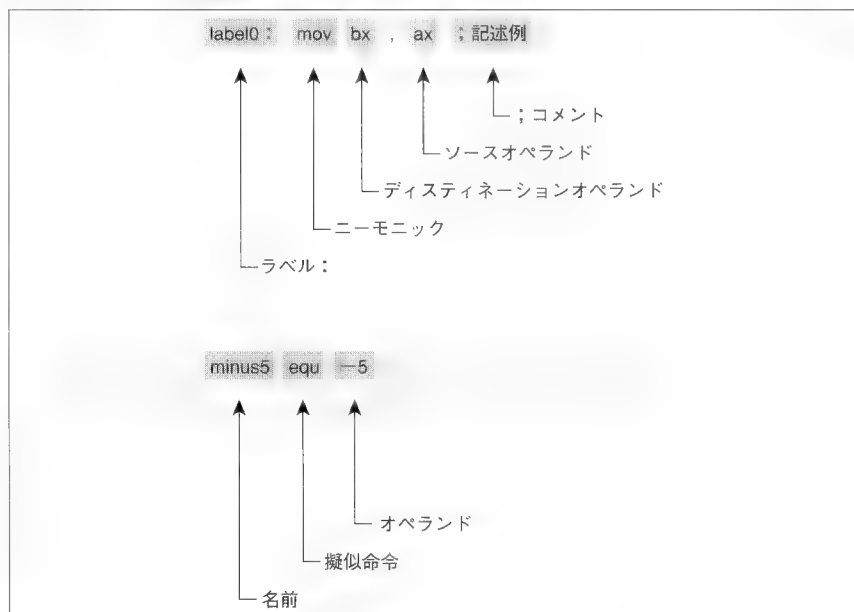
過去の多くのコンピュータのアセンブラ記述形式では，ニーモニックやオペランドは，1行の何桁目に記述する，というように記述位置が決められていました．8086アセンブラでは，ラベル，ニーモニック，オペランド，と書く順番が決まっています．しかし，それをどの桁に書くかは決められていません．どのように間をあけて記述してもかまわないのです．ただし，ラベル，ニーモニック，オペランドのそれぞれは間をあけた書き方はできません．たとえば，label 0:のようにlabelと0の間をあけることはできません．

なお，ラベルやプログラムの名称は，命令やレジスタ名と異なったものを用いるべきでしょう．またオペランドに書くことのできるのは，レジスタ名，変数名，ラベル，式などです．

最後の「;」から後ろはコメントです．命令の意味などを書き込んで，可読性を高めるために用います．;の後ろには，何を書き込んでも機械語変換の際に無視されます．

アセンブリ言語ではもう1つ重要な役割をするものに，疑似命令(ディレクティブ：directive)があります．疑似命令は実際に機械語に変換されることはありませんが，処理系であるアセンブラに定数名や変数名などの情報を伝えたり，配列など連続したメモリ領域を割り当てる役割を果たします．図の例では，minus5という名前が出てきたらそれを−5に置き換えなさい，という意味とな

ります．ここでequ(equate)が疑似命令で，これが機械語命令に変換されることはありません．

図2．3●アセンブラの記述形式

アセンブリ言語の記述での概要を**表2．5**にまとめて示します．1行に1命令を記述します．1行は128文字以内で，特に記述位置の指定のないフリーフォーマットです．たとえばラベルや変数名なども行の最初から書く必要はなく，どこから書き始めてもよいのです．命令を行の最初から書いてもかまいません．そのためにラベルや変数名に用いる名称は英字で始まる必要がありますが，数字を含んでもかまいません．また名称は，命令，疑似命令，レジスタ名とは異なったものを用いる必要があります．これら名称の文字長の制限はありませんが，識別は31文字までとなっています．しかしあまり長い名称を用いると，プログラムの可読性を悪くしますから，適当な長さにするとよいでしょう．

数値を表現する場合，2進数には0101Bのようにその末尾にBを，16進数には0ABCDHのようにHを付けます．16進数がA～Fの文字で始まる場合は，0ABCDHのように先頭に0を付けます．末尾にHやBの文字がない数値は，10

進数と見なされます．

ラベルや命令を書く位置が行の何桁目からと決まっているアセンブラに比べれば，8086アセンブラの記述は楽になっています．

表2．5●記述形式の概要

●１行（128文字以内）に１命令を書く
●英大文字と英小文字は区別されない
●記述位置の指定はなく、ラベルや命令は行のどこから書いても良い
●変数名やラベル名は英字で始まり、数字を含んで良いがその識別は31文字まで
●変数名やラベル名は予約語（命令、疑似命令、レジスタ名）と別にする
●数の表現　２進数　0101B，10進数　123，16進数　0ABCDH

2.4 命令，疑似命令

コンピュータの命令群を命令セットと呼びます．8086の命令セットを，概略分類すると表2．6のようになります．

- ・データ転送命令：レジスタやメモリ間のデータの移動を行う命令で，もっとも多く用いられる命令．
- ・算術演算命令：データに加減乗除などの算術演算を行うための命令．
- ・ビット操作命令：データに論理和，論理積，排他的論理和などの論理演算やシフトローテートを行う命令．
- ・ストリング命令：文字列を扱うための命令で，文字列の移動，演算などを行う．
- ・制御分岐命令：プログラムの流れを制御するための命令で，分岐命令や手続き呼び出し(サブルーチンコール)命令などがある．
- ・プロセッサ制御命令：8086マイクロプロセッサの動きを制御する命令で，割り込み命令などがある．

表2．6●8086命令の分類

	種　別	機　能	命令
1	データ転送命令	データ移動	MOV,XCHG
		スタックデータ移動	PUSH,POP
		入出力データ移動	IN,OUT
		表データ移動	XLAT
		アドレス処理	LEA,LDS,LES
		フラグデータ移動	LAHF,SAHF,PUSHF,POPF
2	算術演算命令	加減算	ADD,ADC,SUB,SBB
		インクリメント、ディクリメント	INC,DEC
		乗除算	MUL,IMUL,DIV,IDIV
		データ変換	CBW,CWD
		データ比較	CMP
		ASCII演算補正	AAA,AAS,AAM,AAD
		10進演算補正	DAA,DAS
3	ビット操作命令	論理演算	NOT,AND,OR,XOR
		論理比較	TEST
		シフト	SHL,SAL,SHR,SAR
		ローテート	ROL,ROR,RCL,RCR
4	ストリング命令	文字列の移動	MOVS,MOVSB,MOVSW
		〃	LODS,STOS
		文字列比較、走査	CMPS,SCAS
		繰り返し処理	REP,REPE/REPZ,REPNE/REPNZ
5	制御分岐命令	無条件、条件付分岐	JMP,J?注2),JCXZ
		ループ処理	LOOP,LOOPE/LOOPZ,LOOPNE/LOOPN
		手続き処理	Z
		割り込み処理	CALL,RET
6	プロセッサ制御命令	フラグ操作	INT,INTO,IRET
		CPU操作	STC,CLC,CMC,STD,CLD,STI,CLI
		No operation	HLT,WAIT,ESC,LOCK

注1）　intel社の命令分類にしたがった
注2）　条件付きジャンプ命令の？部分は、たとえばJZのように条件により変化する

これらの命令には，オペランドを持たない場合，1つのオペランドを持つ場合，2つのオペランドを持つ場合があります．8086の命令をオペランドで分類したものを**表2．7**に示します．

表2．7●8086命令のオペランドによる分類

命令	オペランド
mov	reg/mem,reg
	reg,mem
	reg/mem,imm
	reg/mem,sreg
	sreg,reg/mem
adc	reg/mem,reg
sub	reg,mem
sbb	reg/mem,imm
cmp	
add	
and	
or	
xor	
test	
xchg	reg,reg
	mem,reg
push	reg
pop	mem
	sreg
jmp	short_label
	near_label
	far_label
	regptr16
	memptr16
j?	short_label
call	near_label
	far_label
	regptr16
	memptr16

命令	オペランド
mul	reg
imul	mem
div	
idiv	
inc	
dec	
neg	
not	
lea	reg16,mem16
lds	reg16,mem32
les	reg16,mem32
shr	reg/mem,1
sar	reg/mem,cl
shl	
sal	
rcr	
ror	
rcl	
rol	
in	acc,imm8
out	acc,dx
int	imm8

命令	オペランド
cbw	
cwd	
aaa	
aas	
aam	
aad	
daa	
das	
lahf	
sahf	
pushf	
popf	
clc	
stc	
cmc	
movs	
lods	
stos	
nop	
ret	
into	
halt	
wait	

オペランド・タイプの例

reg	8/16ビット汎用レジスタ
sreg	セグメント・レジスタ
imm	0-ff/0-ffffの範囲の定数
mem	8/16ビット・メモリ・ロケーション
short_label	−128〜＋127バイトの範囲のラベル
near_label	現在のセグメント内のラベル
far_label	他のセグメントのラベル
regptr16	制御を移そうとするコード・セグメントのオフセットを有するレジスタ
memptr16	制御を移そうとするコード・セグメントのオフセットを有するメモリ

マイクロソフトのマクロアセンブラの疑似命令には，セグメントの定義に関して次に示す2つの記述形式があります．

(1) 完全なセグメント定義

(2) 簡略化セグメント定義

(1)の定義が複雑なため，(2)の定義が考えられました．

完全なセグメント疑似命令の例を**図2．4 a)**に示します．assumeはアセンブラの処理系に情報を伝えるための疑似命令で，

```
assume ds:mydata          (セグメント：セグメント名)
```

これから定義されるセグメントをあらかじめ処理系に伝える働きをします．各セグメントは，次のようにsegment疑似命令で始まり，ends疑似命令で終わります．

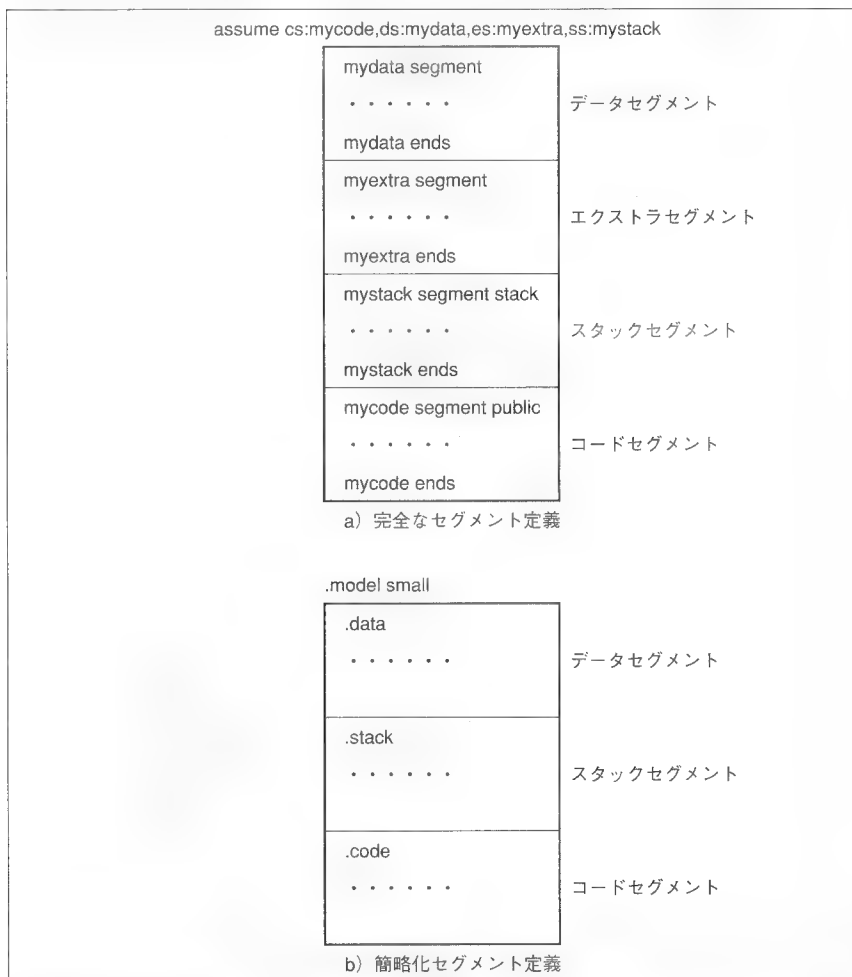
```
┌ セグメント名  segment
│ ・・・・
└ セグメント名  ends
```

セグメント名は最初と最後で同じものを用いる必要があります．セグメント名はユーザが自由に付けることができますが，処理系には定義されたセグメントが，コードセグメントなのかデータセグメントなのかは理解できません．そこで定義したセグメントが，実際にどのセグメントであるかを処理系にわからせるのが，assume疑似命令です．

完全なセグメント定義は記述が複雑なため，簡略化セグメント定義が生まれました．簡略化セグメント定義では.dataなどの簡略化セグメント疑似命令を用います．**図2．4 b)**に簡略化セグメント定義の例を示します．最初に.model smallでメモリモデルを指定します．データセグメントは.data，コードセグメントは.code，スタックセグメントは.stackで定義し，各セグメント終了の疑似命令は付けません．

上記のように簡略化セグメント定義では，簡単にセグメント定義ができます．

よって本書では，簡略化セグメント定義を用いてプログラムし，必要に応じて完全なセグメント定義を用いることにします．

図2．4●セグメント定義

2.5 アドレシングモード

アセンブラでデータを読み書きするときのデータアドレスの指定の仕方を，アドレシングモードと呼びます．データは，レジスタ上にある場合，メモリ上に

ある場合，あるいは定数として命令自身に付属している場合などがあります．データがメモリ上にある場合はメモリアドレシングモードと呼ぶことがあります．単にアドレシングモードと呼ぶ場合は，一般にこのメモリアドレシングモードを指すことが多いのですが，8086ではレジスタの指定も含めてアドレシングモードと呼んでいます．

8086のアドレシングモードを**表2．8**に示します．8086のアドレシングモードは複雑で，次の8種類に分類されます．

- **イミディエート**
- **レジスタ**
- **直接メモリ**
- **レジスタ間接**
- **インデックス**
- **ベースド**
- **ベースドインデックス**
- **ストリング**

8086の命令は，mov ax,memのように，多くの場合2つのオペランドを持ちます．この2つのオペランドに対してそれぞれデータの場所を指定する必要があります．ただし8086では，第1，第2オペランドが共にメモリ上のデータを指定することはできません．イミディエートを除き，一方がメモリならば他方は必ずレジスタを指定することになります．

イミディエート命令は，定数データを扱うもので，

```
        mov         ax,1234H
```

のような場合です．この命令の機械語では，命令自身に定数データが付属した形となります．厳密にいえば，第1オペランドのアドレシングモードはレジスタで，第2オペランドがイミディエートということになります．8086ではオペランドが2つの場合，一方は必ずレジスタになりますから，レジスタ以外のモードがある場合はそれを用いて表現します．

レジスタモードでは，mov ax,bxのようにレジスタの中にデータがあることに

なります．直接メモリモードでは，

```
mov ax,[100H]
mov ax,mem                    ; memは変数名
```

のように直接メモリアドレスを指定します．前述のように8086では，2つのオペランドが共にメモリを指定することはできません．これは直接，間接を問わず指定できません．なお例外的に，ストリング命令では，メモリからメモリへのデータ移動が可能ですが，この命令にオペランドは存在しません．

表2．8●アドレシングモード

	アドレシングモード	対象	概要	例
1	イミディエート	定数	定数が命令に付属	mov ax,1000h
2	レジスタ	レジスタ	レジスタの指定	mov ax,bx
3	直接メモリ	メモリ	アドレスを直接指定	mov ax,[100H]
4	レジスタ間接		アドレスをレジスタで間接指定	mov ax,[b x]
5	インデックス		アドレスをインデックスレジスタで間接指定	mov ax,[si]
6	ベースド		アドレスをベースポインタで間接指定	mov ax,[bp]＋100h
7	ベースドインデックス		アドレスをインデックスレジスタとベースポインタで間接指定	mov ax,[bp+si]＋100h
8	ストリング		文字列をインデックスレジスタで間接指定	movsb

表2．8中の4～7で，一方のオペランドはレジスタを用いて間接的にメモリアドレスを指定しています．この間接アドレスについては次章で詳しく説明しますが，簡単に言えば，レジスタの中に，データのあるメモリアドレスが書き込まれているということです．

プログラムをROM化するときなど，文字定数をコードセグメントに置きたい場合があります．データをアクセスするときは当然データセグメントになりますが，アクセスするセグメントを変更する1バイトの特殊な命令，セグメントオーバライドプリフィックスがあります．これを付けることで，暗黙のうちに決められているセグメントを変更することができます．

なぜこのようにアドレシングモードが複雑なのでしょうか．これは，8086が高級言語(たとえばインテル社のPL/M)などで記述されたプログラムのオブジェクト(機械語)を効率よく作り出すために設計されたことに起因しています．たと

えば配列のアクセスにはインデックスアドレシングが，文字列などの処理にはストリングが，サブルーチンなどで用いられた後には解放されるローカル変数のためにベースドアドレシングが用いられます．

アドレシングモードがたくさんあるため，プログラムを作る際にどのアドレシングモードを使えばよいのかは大いに悩むところです．8086に慣れるまでは，単純化してアドレシングモードを用いてみたらどうでしょうか．

とにかくデータはデータセグメントにだけ置くことにして，次のような単純なアドレシングモードだけを使用します．すなわち定数はイミディエート(これはデータセグメントではなくコードに付属することになりますが)，変数に相当するデータは直接アドレシング，配列に相当するデータはインデックスアドレシングを用いるというように単純化してしまうわけです．スタックデータの参照にはPUSH，POP命令でアクセスすることにし，とりあえずBPはスタックデータの参照には使用しないことにします．

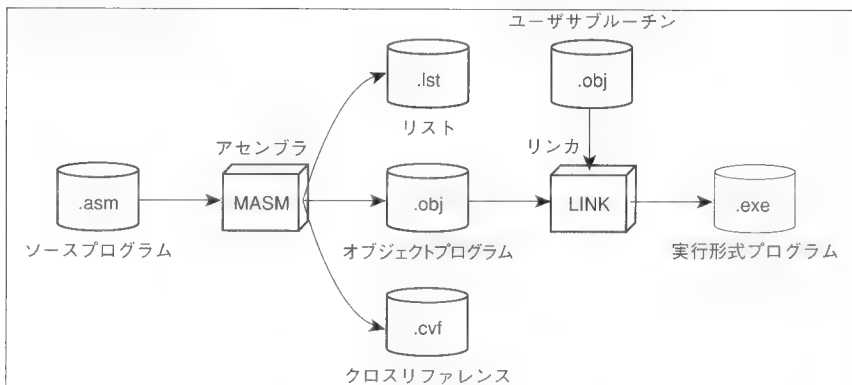
筆者は，8086/88用にPASCALコンパイラを開発して，8086/88を用いた制御プログラムなどはこのコンパイラを用いて記述していました．このコンパイラで生成するオブジェクトコードは，上記3つのアドレシングモードに，スタック上のデータをアクセスするためにBPを用いたベースド，ベースドインデックスを加えただけで可能となっています．実際，上記のような単純化した5つのアドレシングモードだけでかなりのアセンブラプログラムが書けるのです．

2.6 アセンブル，リンク，実行

アセンブリ言語で記述したソースプログラムは，これをアセンブラを用いて機械語に変換し，さらにリンカ(リンケージエディタ)を用いて実行形式のプログラムを作成する必要があります．この実行形式のプログラムを作成することによって，プログラムをMS-DOS上で実行できるようになります．すなわち，プログラムを走らせるためには**図2．5**に示すように，

アセンブル→リンク

の2段階の処理が必要なのです．

図2．5●アセンブル→リンク

アセンブラにはマイクロソフト社のMASM，ボーランド社のTurbo Assembler，多摩ソフトウエア社のLight Macro Assemblerなどがあります．マイクロソフト社のMASMは，マクロ命令が記述可能なアセンブラで，現在はDDK(device driver kit)にリンカLINK，統合エディタQEDITORと共に含まれています．DDKはマイクロソフト社のホームページからダウンロードできますが，これについては付録で説明します．

ここでは，DDKのマクロアセンブラ，リンカを用いて実行形式のプログラムを作成し，実行する場合について説明します．これについては下に示す2つの方法があります．

(1)MS-DOS上で，アセンブル，リンク，実行する．

(2)Windows上で，DDKのQEDITORを用いてアセンブル，リンク，実行する．

(1)の方法を用いるには，最初にWindowsからMS-DOSに移行する必要があります．WindowsからMS-DOSに移行するには図2．6に示すように2つの方法があります．

①**DOS窓を開く**(Windows98の場合)

Windowsのスタートメニュー→プログラム→MS-DOSプロンプト

②**Windowsを終了してMS-DOSで再起動する**

Windowsのスタート→Windowsの終了→MS-DOSモードで再起動する．

①の方法では，エディタでプログラムを修正しながら，アセンブル，リンクができます．Windows上でDOS窓を開いての処理となるため，プログラムの実行に当たって，入出力などに制約がでる場合があります．

図2．6●MS-DOSへの移行方法

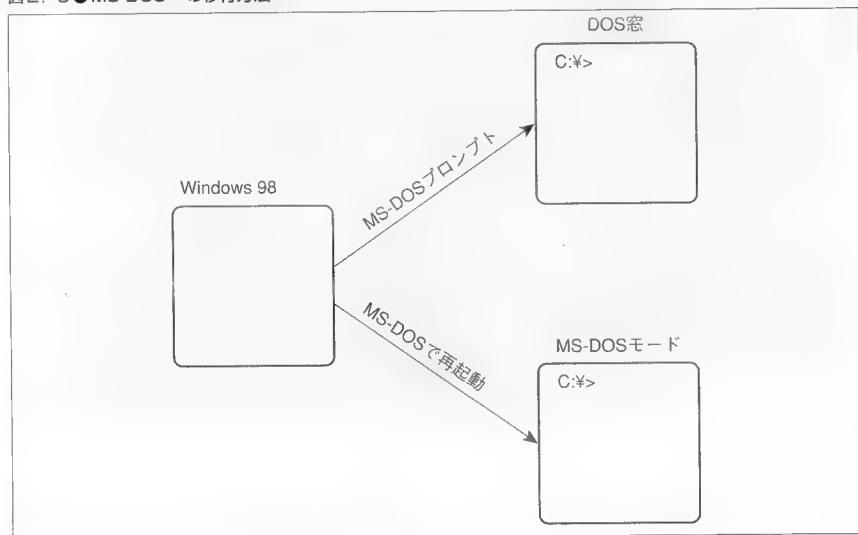



図2．7●MS-DOSプロンプトへの移行

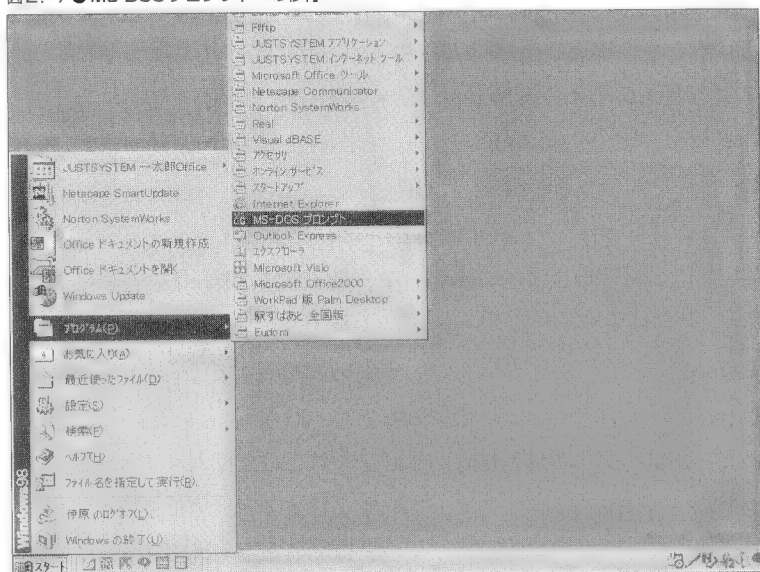
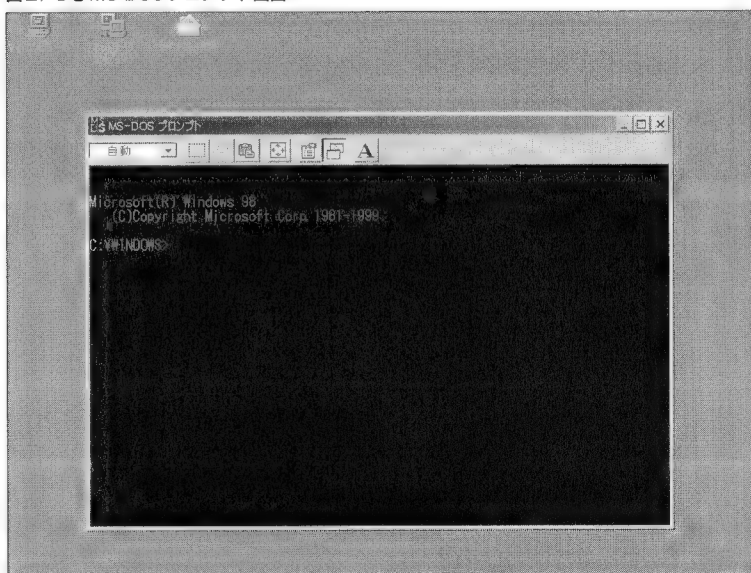



図2．8●MS-DOSプロンプト画面

■MS-DOSからのアセンブル，リンク，実行

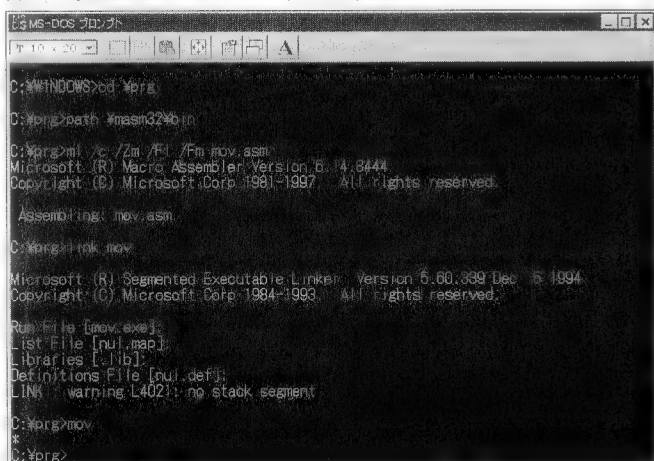
次に示すように，マクロアセンブラ(ml.exeと名付けられている)とリンカ(link.exe)はドライブC:のmasm32¥binディレクトリにあり，アセンブラのソースプログラムはドライブC:のprgディレクトリに保存してあるものとします．

・ソースプログラム	**exmpl.asm**	**C:¥prg**ディレクトリ
・アセンブラ	**ml.exe**	**C:¥masm32¥bin**ディレクトリ
・リンカ	**link.exe**	**C:¥masm32¥bin**ディレクトリ

MS-DOSのコマンドラインから次のコマンドを入力します．

C:¥>cd ¥prg	ディレクトリの変更
C:¥prg>path %path%; ¥masm32¥bin	ディレクトリにパスを通す
C:¥prg>ml /c /Zm /Fl /Fm exmpl.asm	ソースプログラムのアセンブル
C:¥prg>link exmpl	オブジェクトファイルのリンク
C:¥>exmpl	プログラムの実行

なお，マクロアセンブラがあるディレクトリにpathを通すようautoexec.batファイルを書き換えれば，pathコマンドは必要なくなります．

図2.19●MS-DOSでのアセンブル，リンク，実行

■WindowsからQEDITORによるアセンブル，リンク，実行

次にDDKのQEDITORを用いてWindows上でアセンブル，リンク，実行する方法を説明しましょう．

下に示すように，QEDITORはドライブC:のmasm32ディレクトリにあり，アセンブラのソースプログラムexmpl.asmはドライブC:のprgディレクトリにあるものとします．

C:¥prgディレクトリ	**exmpl.asm**	**ソースプログラム**
C:¥masm32ディレクトリ	**qeditor.exe**	**エディタ**
C:¥masm32¥binディレクトリ	**ml.exe**	**アセンブラ**
C:¥masm32¥binディレクトリ	**link.exe**	**リンカ**

Windowsのスタートメニューのファイル名を指定して実行から，

C:¥masm32¥qeditor　　　　**DDKのQEDITORを起動**

図2. 10に示すようにQEDITORウィンドウのメニューで,

①ソースプログラムのオープン

[File]メニュー → [Open]を選択，ソースプログラムを選ぶ.

②アセンブルとリンク

[Project]メニュー → [Assemble & Link]を選択.

③実行

[Project]メニュー → [Run Program]を選択.

この操作で見かけ上WindowsからQEDITORを通して，アセンブル，リンク，実行ができます．実際にはQEDITORを通して，MS-DOS上でこれらの処理が行われており，処理結果などはテキストファイルに保存されて表示されます．アセンブル，リンク，実行した例を図2. 11に示します.

図2. 10●QEDITOR

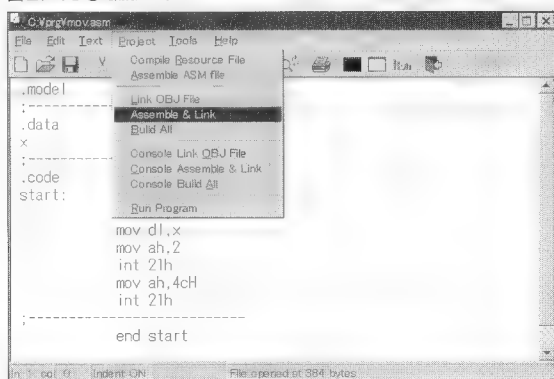
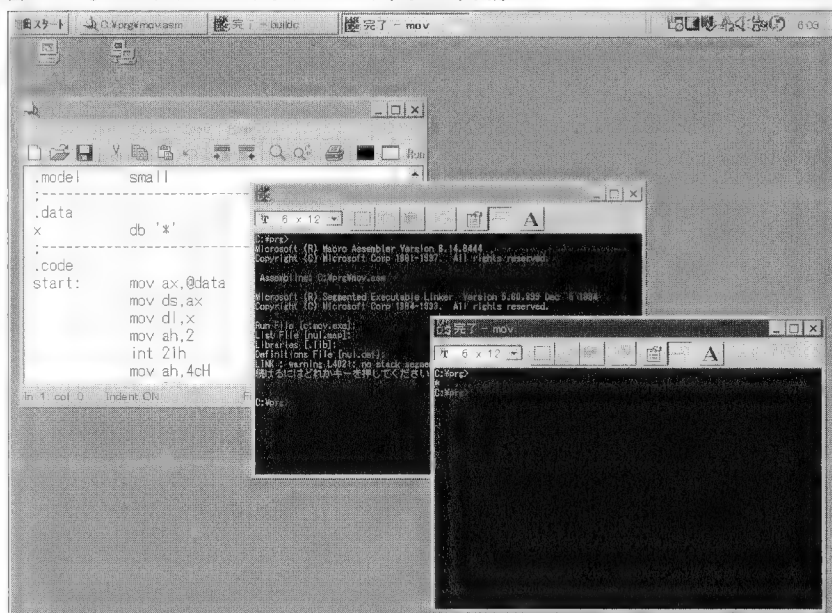



図2．11●DDKのQEDITORによるアセンブル，リンク，実行

演習問題2

1. 2進数，10進数，16進数の変換について，次の問いに答えよ.

(1) 次の10進数を8bitの2進数に変換せよ.

①65

②85

③170

④255

(2) 次の10進数を2の補数(8bit)で表現せよ.

①－1

②－63

③－127

④－128

(3) 次の2進数を10進数に変換せよ．ただし，符号のあるデータとする.

①01101001

②10000001

③11110000

④11111111

(4) 次の2進数を16進数に変換せよ.

①01011010

②11010011

③11100010

④01111111

2. 次の問いを，具体的に説明せよ.

(1) mov bx,ax と mov ax,bx 命令を実際にアセンブルしてそのコード(機械語)を比較せよ.

(2) データセグメントをメモリの80000Hから始めるには，レジスタをどのように設定するか.

(3) データセグメントとエクストラセグメントの先頭を，同一のC0000H番地にするには，レジスタをどのように設定するか.

(4) スタックセグメントを30000Hに，スタックポインタSPをスタックボトムに設定するにはどうするか.

(5) アセンブラの疑似命令の役割はどのようなものか.

第 3 章

データの転送命令

本章では，データの転送に関する命令を具体的に説明します．データ転送命令は，アセンブリ言語プログラムではもっとも頻繁に用いられる命令です．

アセンブリ言語の学習には，実際にプログラムを記述して，コンピュータ上で実行することが重要です．例題や演習問題を，積極的にプログラミングし，実際に走らせてください．処理結果はMS-DOSシステムコールを用いてディスプレイに表示し，結果が正しいかの確認を行います．

表3．1に示すように，データ移動命令，スタック命令，アドレス転送命令，入出力命令，フラグ転送命令があります．データ転送に関しては，単純にレジスタからレジスタへ，レジスタからメモリへ，メモリからレジスタへデータを移動する場合と，互いのデータを交換する場合があります．入出力命令とフラグ転送命令についてはこの章では取り上げず，後の章で説明します．プログラム例の記述は，簡略化セグメント定義を用い，データはデータセグメントに，コードはコードセグメントに配置します．さらに必要に応じて，完全なセグメント定義の例も示します．

表3．1●データ転送命令一覧

分類	命令	機能	
データ移動	MOV	move byte or word	データ転送
	XCHG	exchange byte or word	データ交換
	XLAT	translate byte	表データ転送
スタック	PUSH	push word onto stack	スタックへ保存
	POP	pop word off stack	スタックから復元
アドレス転送	LEA	load effective address	実効アドレスのロード
	LDS	load pointer using DS	ポインタアドレス（DS）のロード
	LES	load pointer using ES	ポインタアドレス（ES）のロード
入出力	IN	input byte or word	データ入力
	OUT	output byte or word	データ出力
フラグ転送	LAHF	load AH register from flags	スタックからAHへフラグを復元
	SAHF	store AH register in flags	スタックへAHからフラグを保存
	PUSHF	push flags onto stack	フラグをスタックへ保存
	POPF	pop flags off stack	フラグをスタックから復元

アセンブリ言語の学習には，実際にプログラムを記述して，コンピュータ上で走らせることが重要です．本書では，常に読者が簡単なプログラムを記述して，確認することを推奨します．章末の演習問題や11章『基本プログラミング』などで積極的にプログラムを実行をしてみてください．

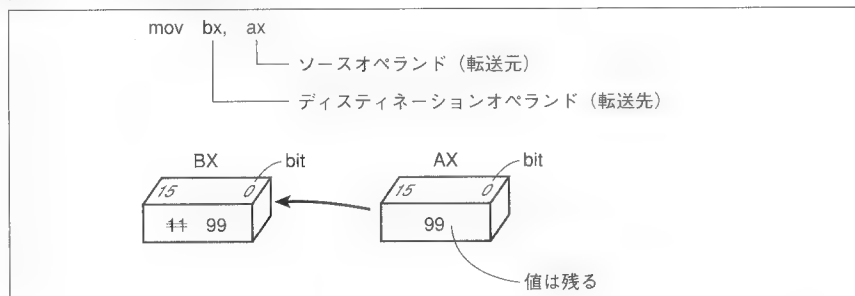
3.1 データ移動命令

汎用的なデータの転送に関連する命令で，レジスターレジスタまたはレジスターメモリ間でデータを転送します．基本的に，転送元であるレジスタまたはメモリ内のデータもそのまま残ります．ニーモニックはmovが用いられ，次の形式となります．

```
mov    destination,source
```

ここで，destinationは転送先で，sourceは転送元を指定します．destinationとsourceはレジスタかメモリアドレスが可能で，sourceについては定数(immediate)も可能です．メモリアドレスは変数名として直接指定するか，レジスタを介して間接的に指定することもできます．8086では，ソースとディスティネーションの両方をメモリアドレスとすることはできません．イミディエート命令を除いて，必ずソースとディスティネーションのいずれかはレジスタでなければならないのです．

図3．1にmov bx,ax命令の例を示します．この命令ではAXレジスタにあるデータをBXレジスタへ転送します．初期値としてAXには99が，BXには11があるとします．このときmov bx,axを実行すると，AXにあったデータ99がBXに移されます．したがってBXの値は11から99に書き換えられます．しかしAXのデータ99は消えることなくそのまま残ります．データ転送といっても，コピーをするように，転送元のデータはそのまま残ります．

図3．1●mov bx,ax命令の例

リスト3．1●移動命令の使用例

```
1:  .code                   ;コードセグメントの先頭
2:      mov ax,bx           ;bxの内容をaxへ
3:      mov al,bh           ;bhの内容をalへ
4:      mov ah,bl           ;blの内容をahへ
5:      mov cl,x            ;バイト変数xの内容をclへ
6:      mov ax,w            ;ワード変数wの内容をaxへ
7:      mov ax,3            ;直接データ3をaxへ
8:      mov x,5             ;直接データ5を変数xへ
9:  .data                   ;データセグメントの先頭
10: x   db  ?               ;byte変数x
11: w   dw  ?               ;word変数w
```

リスト3．1にデータ移動命令の使用例を示します．8086ではコードはコードセグメント，データはデータセグメントにあることが基本となっています．本書のプログラムでは，コードセグメントやデータセグメントを明確に定義します．行1:の.codeはコードセグメント定義の簡略化セグメント疑似命令で，ここからコードセグメントが始まることを示しています．行9:の.dataはデータセグメント定義の簡略化セグメント疑似命令で，ここからデータセグメントが始まることを意味します．

行10:のxは変数で，db(define byte)疑似命令によってバイトデータとなり1バイトのメモリ領域が用意されます．?はデータを初期化しないことを示しています．コンピュータでの変数はメモリに名前を付けたもので，データを記憶できる場所を意味します．行11:のwは同じく変数で，dw(define word)疑似命令によってワードデータとなり2バイトのメモリ領域となります．これら疑似命令は命令として実行されるのではなく，アセンブラ処理系に情報を伝える役割を果たします．

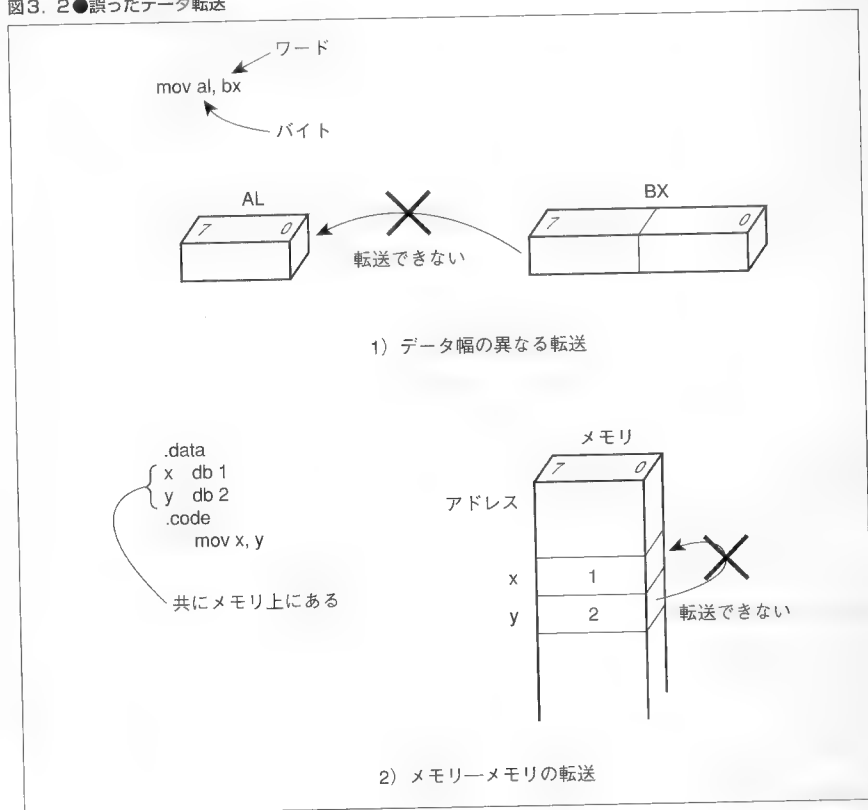
命令の使用例は，行2:～8:で，mov ax,bxではBXレジスタの内容が，AXレジスタへ送られます．前述のようにニーモニックmovの後のオペランドは，データを送られる側(ディスティネーション)を前に，送る側を後ろに記述します．AXレジスタはワード，BXレジスタもワードなので，送り側と受け取る側のデータ幅が一致しています．このようにソースとディスティネーションのデータ幅は一致している必要があります．転送後も，BXレジスタの内容はなくなってし

まうのではなく，元のまま残ります．

なお，次のデータ転送は，例外を除いて誤りとなります．

- データ幅の異なる転送(バイトからワードへ　など)
- メモリ―メモリ間の転送

図3．2の1）に示すように，ソースとディスティネーションのデータ幅が異なっているmov　al,bxのような記述は誤りとなります．ALはバイト，BXはワードレジスタであるために，データ移動ができないのです．また2）に示すように，mov　x,yのようなメモリ―メモリ間のデータ転送も誤りとなります．8086では例外を除いて，メモリからメモリへのデータ転送はできません．

図3．2●誤ったデータ転送

行3:のmov al,bhでは，バイト単位のデータ移動で，BHレジスタ(BXレジスタの上位8ビット)の内容が，ALレジスタ(AXレジスタの下位ビット)へ移動します．もちろんBHレジスタの内容はそのまま保たれます．

行5:のmov cl,xでは，メモリのxと名付けられたアドレスの内容をCLレジスタへ移動します．このxはデータセグメント内に設定されています．行6:のmov ax,wではメモリのwと名付けられたアドレスの内容をAXレジスタへ移動します．このときwがdwでなくdbで領域をとってあると，メモリ上の次の番地のデータを含めて移動してしまいます．

行7:のmov ax,3では，定数3を直接AXレジスタへ移動します．このとき定数3の値は，データ領域にとられるのではなくこのmov命令のコードに直接付属します．そこでこのようなアドレシングモードを，直接データ(イミディエート)と呼びます．

行8:のmov x,5は，直接データ5をメモリのx番地へ書き込む命令です．xはdbとバイト変数として領域をとってあるので，この命令では定数5はバイトとして扱われます．これらがバイトかワードかは，アセンブラ処理系が自動的に判断してくれます．

前述したように行9:の.dataは簡略化疑似命令で，この疑似命令以降のデータがデータセグメントに置かれることを宣言します．データがデータセグメントに置かれるのは当然と考えられますが，実はそうでない場合も多く見かけられます．データをコードセグメントへ直接定義してしまうことはかなり頻繁に見かけます．しかし本書では，基本的にコードはコードセグメントに，データはデータセグメントに，スタックはスタックセグメントに置くことにします．そうすることによって，プログラムが簡単になり，わかりやすくなります．

移動命令を簡単に説明したので，ここで実際にプログラムを実行して確かめてみます．まだ紹介していない命令がいくつか出てきますが，とりあえずそんなものかと考えてください．詳しくは後章で説明します．プログラミングはまだ早いと考えるかも知れませんが，実際にプログラムを実行してみるのが，アセンブラ上達の早道なのです．それに，簡単なものでも，プログラムが完成したという達成感も得られます．

リスト3．2に移動命令のプログラム例を示します．MS-DOSから例題プログ

ラムを呼び出して実行し，実行が終了したら，再びMS-DOSへ戻る構成となっています．MS-DOSの呼び出し方などは，後の章で詳しく説明します．ここで実際に知ってほしいのは，プログラムの網掛け部分です．

このプログラムでは，変数xに初期設定された文字*を，ALレジスタへ移動した後，変数yへ移動します．直接xからyへ移動することはできません．その後確認のため，yの値をディスプレイへ表示させます．yの文字を表示するためにMS-DOSシステムコールを用います．MS-DOSシステムコールは，MS-DOSに用意された，データの入出力に関する標準的なプログラム(ファンクション)です．これをシステムコールという形で利用することができます．

リスト3．2はアセンブルリストを示したもので，全体を簡単に説明します．mov命令とdb疑似命令を除けば，概略の理解ができればそれで充分です．左端の1～21までの番号はプログラムの行番号です．その右の0000～0016，0000～0001とあるのはコード(機械語命令)や変数があるメモリアドレスを示しています．行3:の.codeで始まるコードセグメントと行17の.dataで始まるデータセグメントはそれぞれ16進数の0000から始まっています．アドレスの右にあるのが，実際の機械語を16進数で表現したものです．行4:のコードB8 ---- Rは，mov ax,@dataを機械語にしたもので，B8 ----はそれぞれ1バイトを示します．----はアセンブル時にはアドレスは与えられず，実際にプログラムが実行されるときに与えられます．

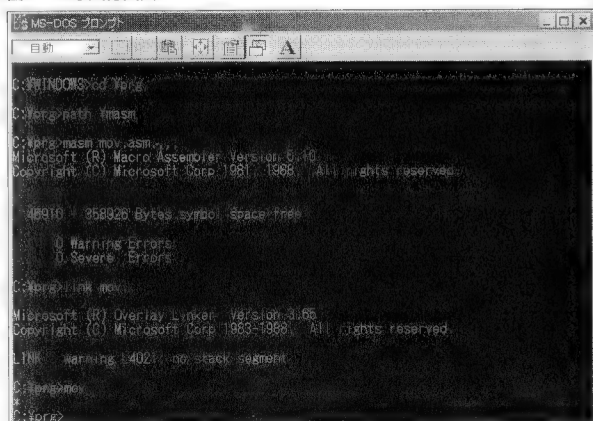
行3:の.codeはコードセグメントの始まりを示しています．行4:にあるstart:とあるのは，このアドレスを示すために付けたラベルです．このようにラベルを付けておくと，このアドレスを参照したい場合，たとえば0000H番地と書く必要はなく，startと書くことによって参照することができます．行4:～5:のmov命令は，MS-DOS上でプログラムが走るときに必要な操作で，データセグメントレジスタDSに，実際のデータセグメントアドレスを与えています．

行7:～8:のmov命令が，実際にデータを移動しているものです．8086では基本的に，メモリからメモリへのデータ移動はできません．このため変数xから変数yへは直接データを送ることができず，変数のx→AL→yという経路でデータを送っています．この場合データの移動とはいっても，送り元のデータはなくなってしまうわけではなく，元のまま残っています．

行10:〜12:はMS-DOSのシステムコールを用いてデータをディスプレイに表示する命令です．行14:〜15:はMS-DOSへ復帰するためのシステムコールです．行4:〜5:と，行10:〜15:のMS-DOSシステムコールは，ここではっきりと理解できなくてもよいのです．これから本書で示していく例題や，演習題を実際に動かすことによって理解することができるようになります．

リスト3．2●移動命令の例題プログラム（xの文字をyへ移動し表示する）

```
行   アドレス コード            プログラム              コメント

1:                             .model small          ;メモリモル small
2:                             ;----------------------------------------
3:  0000                       .code                 ;コードセグメントの始まり
4:  0000 B8 ---- R             start: mov ax,@data   ;データセグメントアドレス
5:  0003 8E D8                        mov ds,ax      ;データセグメントを設定
6:                             ;
7:  0005 A0 0000 R                    mov al,x       ;xのデータをalへ移動
8:  0008 A2 0001 R                    mov y,al       ;alのデータをyへ移動
9:                             ;
10: 000B 8A 16 0001 R                 mov dl,y       ;yのデータをdlレジスタへ移動
11: 000F B4 02                        mov ah,2       ;MS-DOSファンクション2(文字表示)
12: 0011 CD 21                        int 21h        ;MS-DOSシステムをコール
13:                            ;
14: 0013 B8 4C00                      mov ax,4c00H   ;ファンクション4c(OS復帰)
15: 0016 CD 21                        int 21h        ;MS-DOSシステムをコール
16:                            ;------------------------------
17: 0000                       .data                 ;データセグメントの始まり
18: 0000 2A                    x      db '*'         ;文字*で初期化する変数xを定義
19: 0001 00                    y      db ?           ;初期化しない変数yを定義
20:                            ;------------------------------
21:                                   end start      ;startから実行開始


実行例
    C:¥prg>mov
    *
```

図3．3●実行画面

3.2 間接アドレス

間接アドレスとは，直接メモリアドレスを指すのではなく，レジスタに必要なメモリアドレスを書き込んで，データ移動を行うものです．したがって，間接アドレスに用いるレジスタへは，あらかじめメモリアドレスを書き込んでおく必要があります．

なぜこのような複雑なことをするのでしょうか?

メモリ上のデータを処理する場合，連続したメモリアドレスにデータを割り当てることがよくあります．たとえば高級言語での配列を想像してみてください．一連のメモリ領域に割り当てられたデータをアクセスするには，その先頭番地から番地を1ずつ増やしていけば処理が簡単になります．配列の先頭アドレスをレジスタに保存して，このレジスタが示すアドレスのデータにアクセスできるようにすれば，アドレスを1ずつ増やすごとに，次のデータをアクセスすることができます．このような用途のために，間接アドレシングが用意されているのです．

表3．2に間接アドレスに用いることのできるレジスタを示します．間接的にメモリアドレスを指定できるレジスタにはインデックスレジスタSI，DIとベースレジスタBP，BXがあります．その他のレジスタAX，CX，DXは，間接アドレスに用いることはできません．間接アドレスに用いるレジスタによって，デー

タのあるセグメントが異なってきます．

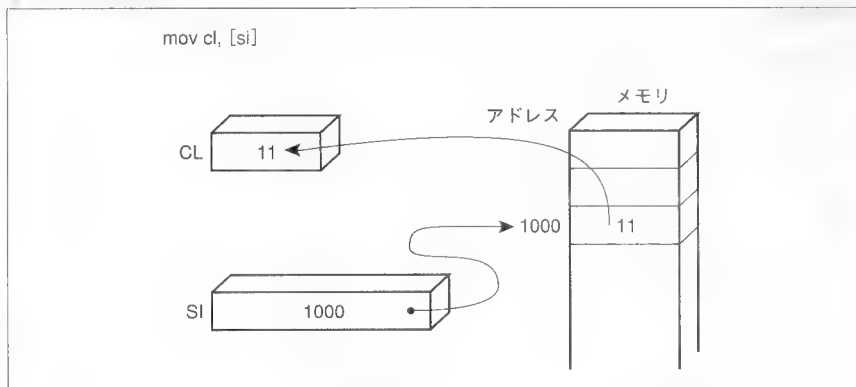
この表はアドレシングがわからなくなったとき役に立ちます．たとえば，データをアクセスする際に直接アドレスを指定した場合や，BPを除くレジスタ(BX,SI,DI)で間接的にアドレシングしたときはデータセグメント(DS)にデータがあることになります．BPを用いて示されるアドレスは，自動的にスタックセグメント(SS)になります．

表3．2●間接アドレスに使用できるレジスタ

レジスタ	データのあるセグメント	レジスタの役割
SI	DS	インデックスレジスタ
DI	DS	インデックスレジスタ
BX	DS	ベースレジスタ
BP	SS	ベースレジスタ

間接アドレスの例を**図3．4**に示します．ここでは間接レジスタとしてSIを用いています．**表3．2**に示したように，SIレジスタが指す領域はデータセグメントです．SIレジスタにはすでに，1000番地のメモリアドレスが与えられており，メモリの1000番地には11というデータがあります．この1000番地のデータを間接的にアクセスする例です．mov cl,[si] 命令によって，間接アドレスを用いて1000番地の内容11(1バイト)がCLレジスタへ転送されます．ここで[SI]は，SIレジスタを用いた間接アドレスであることを示しています．すなわちSIレジスタには，メモリアドレスがあることになります．

もしソースオペランド[SI]に[]がなければmov cl,siとなり，SIレジスタの内容(この場合は1000)を，CLレジスタへ移動するという命令になります．しかし，CLレジスタは8ビット，SIレジスタは16ビットのレジスタですから，mov cl,si と記述することはエラーとなります．

図3．4●間接アドレス

リスト3．3に間接アドレスの移動命令の例を示します．間接アドレスを用いて，変数xの内容を変数yへ移動する例です．xとyはデータセグメント上にあり，xには文字*がデータとして与えられているものとします．

行2:のmov si,offset x 命令で，SIレジスタに変数xの番地を書き込みます．ここで，offsetとはxのアドレスを算出するために用いる演算子で，**図3．5**に示すようにデータセグメントの先頭からxまでの距離を示します．もしoffsetがなければmov si,xとなって，xの内容がSIレジスタへ転送されます(ワード，バイトでエラーとなりますが)．offsetがついている場合は，変数xが置かれるメモリアドレスがSIレジスタへ渡されます．

行3:のmov di,offset yでも，変数yのアドレスがDIレジスタに設定されます．

行4:のmov cl,[si]で，[]で囲まれたsiは間接アドレスを意味します．SIレジスタの内容が示しているメモリ番地(すなわち変数xの番地)のデータをCLレジスタへ移動するという意味です．もしこのsiへの[]がないmov cl,siだった場合，単にSIレジスタの内容をCLレジスタへ移動する，という命令になります．これは，バイトレジスタの内容をワードのSIレジスタへ移動する命令となるため，誤りとなります．

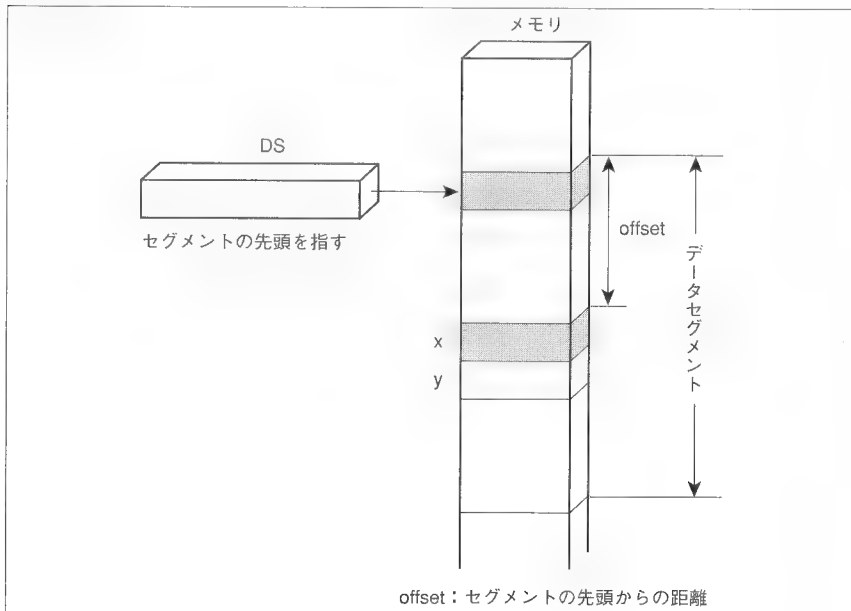
行5:では，このCLレジスタのデータが，DIレジスタが指すメモリ番地へ書き込まれます．これで，間接アドレスを用いて変数xの内容が，変数yへ移動したことになります．行4:と行5:では，mov [di],[si]という書き方ができれば簡単なのですが，8086の命令には，メモリーメモリというオペランドを用いること

ができません．このため，行4:と行5:の2つの命令を用いたわけです．

リスト3．3の例では，x番地の1データをy番地に移動しているだけですが，これでは間接アドレスの有効性が理解できないと思います．たとえば配列のように，メモリの連続したアドレスに割り当てられたデータをアクセスする場合に，間接アドレスは有効なのです．SIやDIが指すメモリ番地を1ずつ順番に増やしていけば，簡単に配列データをアクセスしていくことができます．

リスト3．3●移動命令の使用例(間接アドレス)

```
1:  .code                     ;コードセグメント
2:       mov si,offset x      ;変数xのアドレスをsiへ設定
3:       mov di,offset y      ;変数yのアドレスをdiへ設定
4:       mov cl,[si]          ;番地をsiで間接に指定
5:       mov [di],cl          ;[]は間接にアドレス指定
6:  .data                     ;データセグメント
7:  x    db  '*'              ;byte変数x
8:  y    db  ?                ;byte変数y
```

図3．5●オフセット

間接アドレスを理解するために，前出のプログラムをMS-DOS上で走らせてみます．動作確認のためにMS-DOSシステムコールをして，変数yの内容を表示します．アセンブルしたプログラムリストを**リスト3．4**に示します．プログラムは，変数xの内容を変数yに移してから表示するというものです．これは，間接アドレスを用いていることを除けば，**リスト3．2**で示した処理と同じ処理をしています．

メモリモデルの指定や，データセグメントの設定も**リスト3．2**と同様です．行7:のmov si,offset xでは，変数xのアドレスをSIレジスタへ設定しています．この命令で生成されたコードは，コードセグメントの5～7番地で，BE 0000というコードになっています．最初のBEはmov命令のコードで，次の0000が変数xのアドレスになっています．行20:のxのアドレスをみると，データセグメントの0000番地にあり，コードのアドレスと一致しています．行8:ではDIレジスタに変数yのアドレスを設定しています．ここで，変数yのアドレスとして0001がコードの中に示されています．これは行21:の変数yの実際のアドレスが0001になっているのと一致しています．

このプログラムを実行した例では，MS-DOSのコマンドライン上に移動した文字*が表示されています．

リスト3．4●間接アドレスによる移動のプログラム例（xの文字をyへ間接アドレスで移動，表示する）

```
行  アドレス コード        プログラム              コメント

1:                        .model small         ;メモリモデルsmall
2:                        ;-------------------------------------------
3:  0000                  .code                ;コードセグメントの始まり
4:  0000 B8 ---- R        start: mov ax,@data  ;データセグメントアドレス
5:  0003 8E D8                   mov ds,ax     ;データセグメントを設定
6:                        ;
7:  0005 BE 0000 R               mov si,offset x;変数xのアドレスをsiへ設定
8:  0008 BF 0001 R               mov di,offset y;変数yのアドレスをdiへ設定
9:  000B 8A 04                   mov al,[si]   ;siで間接指定するデータをalへ
                                                移動
10: 000D 88 05                   mov [di],al   ;diで間接指定するアドレスへal
                                                のデータを移動
11:                       ;
12: 000F 8A 16 0001 R             mov dl,y      ;yのデータをdlレジスタへ移動
```

第3章 データの転送命令

```
13: 0013 B4 02                 mov ah,2      ;ファンクション2(文字表示)
14: 0015 CD 21                 int 21h       ;MS-DOSシステムコール
15:                  ;
16: 0017 B8 4C00               mov ax,4c00H ;OS復帰  ファンクション4C
17: 001A CD 21                 int 21h       ;MS-DOSシステムコール
18:                  ;------------------------------------------
19: 0000             .data                   ;データセグメントの始まり
20: 0000 2A          x         db  '*'       ;文字*で初期化する変数xを定義
21: 0001 00          y         db  ?         ;初期化しない変数yを定義
22:                  ;------------------------------------------
23:                            end start     ;startから実行開始


実行例
    C:¥prg>ind
    *
```

間接アドレスを用いる場合，間接アドレスに用いるレジスタへ変数や配列のアドレスを与える必要があります．**リスト3．4**では

```
mov si,offset x
```

という形でSIレジスタへ変数xのアドレスを与えました．これは，アセンブル時に変数xのアドレスがわかっているためにこのような記述ができます．しかし，複雑なアドレシングを用いた場合はこのようにできない場合があります．そのために**表3．1**に示したlea命令があります．この命令はプログラムの実行時に，実効アドレスをレジスタへロードする命令です．たとえば，

```
lea si,x
lea di,y
```

と記述することでこれが可能となります．**リスト3．4**の例では，どちらを用いても同じ結果が得られますが，今後はこのlea命令で実効アドレスを与えることにします．

3.3 アドレシング

ここで，本書で用いる8086のメモリアドレシングについて，整理をしておきましょう．8086のアドレシングモードについては2章で取り上げましたが，アドレシングモードが複雑であるために，**表3．3**に示すような単純化したアドレシングモードを用いることを提案しました．

表3．3●単純化したアドレシングモード

	アドレシングモード	対象	概要	例
1	イミディエート	命令	定数が命令に付属	mov ax,1000h
2	レジスタ	レジスタ	レジスタの指定	mov ax,bx
3	直接	メモリ	アドレスを直接指定	mov ax,x
4	インデックス		アドレスをインデックスレジスタで間接指定	mov ax,[si]
5	ストリング		文字列をインデックスレジスタで間接指定	movsb

それらは，次のようなものです．

- **データはデータセグメントに置く**
- **定数はイミディエート(直接データ)**
- **変数は直接アドレシング**
- **配列はインデックスレジスタによる間接アドレシング**

スタックデータは後に出てくるpush，pop命令でアクセスすることにします．とりあえず，BPはスタックデータの参照には使用しないことにします．基本的なプログラムはこれで充分なはずです．その他，構造体に相当するデータなどについては後の章で取り上げることにします．

リスト3．5に単純化したメモリアドレシングモードをまとめてあります．行10:のデータセグメントのxは文字*を初期値とした変数です．xは直接アドレシングでアクセスすることにします．行11:のyは初期化しない変数で，これも直接アドレシングでアクセスします．行12:のzは100の要素を持つ配列で，dup(?)によって初期化されないことになります．z　db 100 dup(?)については注意が必要です．単にz　db 100と記述すると，zは初期値100を持つ1バイト

の変数となります．dup(?)が付くことによって，zは配列となります．

行2:と行3:では，変数xとyを直接アドレシングで指定しています．行4:ではBLレジスタをクリアするため，イミディエート(直接データ)として命令の機械語に直接データを付属させます．行5:では配列zの先頭アドレスをDIレジスタへ設定するため，zにoffsetを付けてアドレスとしています．実際の機械語としては，このアドレスがイミディエートデータとして用意されることになります．行6:では配列zの最初の要素にALの内容を移動するため，DIレジスタを間接アドレシングに用いています．

リスト3．5●単純化したアドレシングモード

```
1:  .code
2:      mov al,x            ;直接アドレシング(byte)
3:      mov y,bx            ;直接アドレシング(word)
4:      mov bl,0            ;イミディエート(byte)
5:      mov di,offset z     ;イミディエート(word)配列zのアドレス
6:      mov [di],al         ;間接アドレシング(byte)
7:      lea si,w            ;変数wのアドレスのロード
8:      mov y,[si]          ;間接アドレシング(word)
9:  .data
10: x   db  '*'             ;byte変数  文字*で初期化
11: y   dw  ?               ;word変数  初期化せず
12: z   db 10 dup(?)        ;byte配列  領域10byteで初期化せず
13: w   dw 50 dup(?)        ;byte変数  領域50wordで初期化せず
```

3.4 配列のアドレシング

ここでは，配列データのアドレシングについて述べます．ここの説明は入門者には少し難しいかもしれません．もし難しいと感じたなら，ここは読み飛ばして，必要になったときに戻って理解するとよいでしょう．

配列データのアドレシングにはいくつかの方法があります．**リスト3．6**に配列データの要素の値を移動する例を示します．データセグメントにバイト配列bとワード配列wがあるものとします．行2:～3:の

```
mov si,offset b
mov al,[si]
```

では，最初にインデックスレジスタSIに配列bのアドレスを与えます．offset bとしてbのアドレスを指定する場合，命令としてはイミディエートとしてアセンブル時にbのアドレスが計算されます．このSIを間接アドレスとして，ALに配列bの要素の値を移動しています．bの他の要素をアクセスする場合はinc，dec命令を用いてアドレスを増減させます．

offsetと同じ効果を与える例が行4:～5:の

```
lea si,b
mov al,[si]
```

です．lea命令では，SIレジスタへbの実効アドレスを与えています．このアドレスはアセンブル時ではなくプログラムの実行時に計算されることになります．データの移動が，SIを間接アドレスとして用いるのは前例と同じです．bの他の要素をアクセスする場合に，inc，dec命令を用いてアドレスを増減させる必要があるのも前例と同じです．アドレスを計算するのに，mov命令でoffsetを用いても，lea命令を用いてもどちらでも同じ効果となります．データセグメントの宣言が複数行われているような場合を想定すると，lea命令を用いた方が安全でしょう．

2つの例では，前述した簡略化したアドレシングモードを用いています．次に，少し複雑なアドレシングの例を取り上げましょう．行6:～7:ではインデックスレジスタは，配列bのアドレスではなく要素番号を示すように用います．

```
mov si,0
mov al,b[si]
```

最初にSIへbの要素番号0を与えています．これは前2例と異なりアドレスではなく，0から始まる番号となります．次のmov命令のソースオペランドで

b[si]とあるのは，配列bの要素をSIが指すということになります．これで指定された配列bの要素の値が，ALへ移動することになります．他の要素をアクセスする場合は，inc，dec命令を用いて要素番号を増減させます．配列へのアドレシングとしては，前2例に比べ少し複雑になります．本書の単純化したアドレシングでは，この方法を用いません．

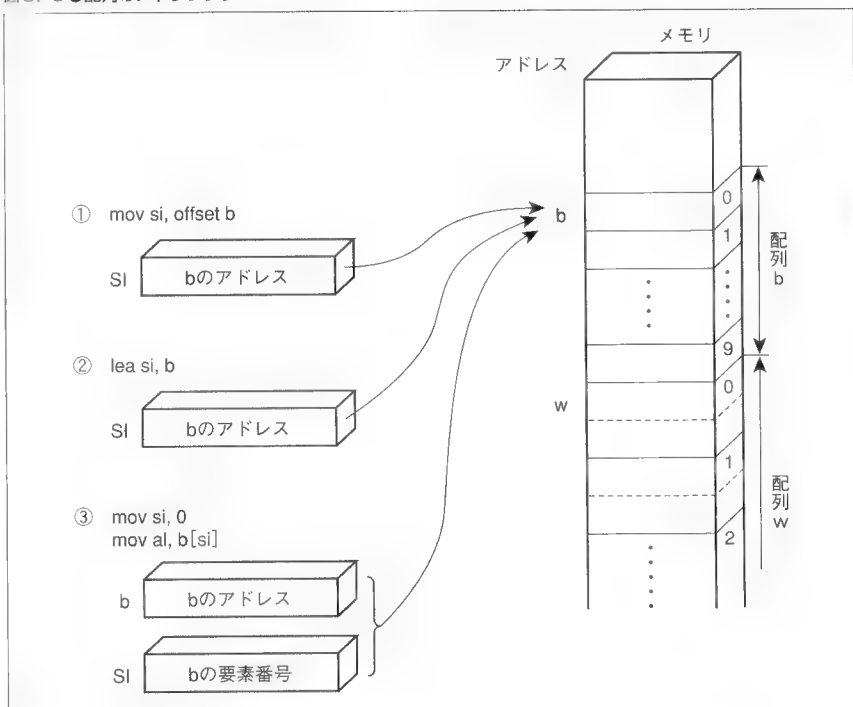
図3．6にそれぞれのアドレシングの違いを示します．

①はoffsetで計算したbのアドレスをSIレジスタが持つ場合です．②はleaで計算したbのアドレスをSIレジスタが持っていて，①の場合と同じくSIを間接アドレスのレジスタとして用います．③では，bのアドレスはmov　al,b[si]命令自身に付いており，このアドレスと要素番号を持つSIとの2つを用いて実効アドレスが計算されます．

行8:～9:はワードデータのアクセス例です．ここでも，DIとSIはワード配列wの要素を示すために用いています．

リスト3．6●配列データのアドレシング

```
1:  .code
2:      mov si,offset b         ;配列bのアドレスをsiに設定
3:      mov al,[si]             ;配列bの要素をalへ移動
4:      lea si,b                ;配列bのアドレスをsiに設定
5:      mov al,[si]             ;配列bの要素をalへ移動
6:      mov si,0                ;siへ要素番号0を設定
7:      mov al,b[si]            ;配列bのsiが指す要素をalへ移動
8:      mov w[di],cx            ;cxを，配列wのdiが指す要素へ移動
9:      mov dx,w[si]            ;配列wのsiが指す要素をdxへ移動
10: .data
13: b   db 10 dup(?)            ;byte配列　領域10byteで初期化せず
14: w   dw 50 dup(?)            ;word配列　領域50wordで初期化せず
```

図3．6●配列のアドレシング

配列データを表示するプログラムを**リスト3．7**に示します．文字列で初期化した配列xを表示するプログラム例です．配列のアクセスのためカウンタを用いていますが，これらに関する命令は後の章で説明をします．ここでは，網掛けした部分を理解してもらえれば充分でしょう．

行19:～23:でデータの定義をしています．配列xは15文字'macro assembler'で初期化しています．変数ctrは配列の要素をカウントする変数で，?では初期化を行いません．

行7:～14:までが実際の処理部分で，最初にカウンタに用いる変数ctrに配列の要素数を与えています．次にlea si,xでSIレジスタに配列xのアドレスを与えています．ラベルdspは，配列の要素すべてを処理するループのために定義しています．この行9:～14:までの命令が，ctrが0になるまで繰り返し実行されます．mov dl,[si]でDLレジスタに配列の要素の内容をロードし，1文字表示するMS-DOSシステムコールを行います．その後，次の配列要素参照のためinc si

でSIレジスタをインクリメント(1増)し，さらにdec ctrでカウンタctrをディクリメント(1減)しています．行14:のjnz dspでは，前のdec命令でctrの値が0になっていなければ，ラベルdspへ飛びます．jnzは条件付きジャンプ命令で詳しくは後の章で説明します．

リスト3．7●配列データのプログラム（配列xの文字を表示する）

```
 行  アドレス コード             プログラム                コメント

 1:                          .model small          ;メモリモデルsmall
 2:                          ;-----------------------------------------
 3: 0000                     .code                 ;コードセグメント
 4: 0000 B8 ---- R           start: mov ax,@data   ;データセグメントアドレス
 5: 0003 8E D8                      mov ds,ax      ;データセグメントを設定
 6:                          ;
 7: 0005 C6 06 000F R 0F            mov ctr,15     ;カウンタctrへデータ数15を設定
 8: 000A 8D 36 0000 R               lea si,x       ;レジスタsiに配列xのアドレスを設定
 9: 000E 8A 14               dsp:   mov dl,[si]    ;配列yの要素をdlレジスタへ移動
10: 0010 B4 02                      mov ah,2       ;文字表示ファンクション番号2
11: 0012 CD 21                      int 21h        ;MS-DOSシステムコール
12: 0014 46                         inc si         ;アドレスを1増
13: 0015 FE 0E 000F R               dec ctr        ;データカウンタctrを1減
14: 0019 75 F3                      jnz dsp        ;カウンタが0でないなら
15:                          ;
16: 001B B8 4C00                    mov ax,4c00H   ;OS復帰ファンクション4C
17: 001E CD 21                      int 21h        ;MS-DOSファンクションを呼び出し
18:                          ;------------------------------
19: 0000                     .data                 ;データセグメント
20: 0000 6D 61 63 72 6F 20   x      db 'macro assembler' ;初期化する配列xを定義
21:      61 73 73 65 6D 62
22:      6C 65 72
23:      000F 00             ctr    db ?           ;データカウンタ
24:                          ;------------------------------
25:                                 end start      ;startから実行開始


実行例
    C:¥prg>array1
    macro assembler
```

3.5 データの交換

プログラムを書いていく場合，単にデータを移動するだけでなく，レジスタとメモリのデータ交換をしたいときがあります．mov命令を用いて，AXレジスタとBXレジスタの内容を交換する例をリスト3．8のa)に示します．この場合，CXレジスタをAXレジスタの一時保存場所として用いています．AXをCXに退避した後，AXにBXの内容を移動，BXにCXの内容を移動して，結果的にAXとBXの交換ができることになります．ここではCXをデータ保存のために用いているため，元々あったCXの値は消えてしまいます．スタックなどを作業場所として用いることもできますが，レジスタやメモリに影響を与えることなくデータの交換ができれば，それの方がいいでしょう．

そのために用意されたのが，xchg(exchange)命令です．xchg命令は2つのオペランドのデータを交換しながら，他のレジスタやメモリの内容に影響を与えません．AXとBXレジスタの内容をxchg命令を用いて交換する例を，リスト3．8のb)に示します．単に，xchg ax,bxと書くことによって，AXとBXの内容が交換されます．

xchg命令は，メモリとレジスタの内容を交換することもできます．メモリとレジスタの内容を交換する例を次に示します．

```
xchg   x,ax
```

メモリアドレスxをディスティネーションオペランドに記述するのが基本形ですが，xchg ax,xの順にオペランドを記述しても問題はありません．

リスト3．8●データ交換の使用例

```
1:  mov cx,ax               ;axの内容をcxに一時保存.cxの初期値は消える
2:  mov ax,bx               ;bxの内容をaxに移す
3:  mov bx,cx               ;cxに保存しておいたaxの内容をbxへ移す

        a)mov 命令を使用
```

```
4:  xchg ax,bx                  ;axとbxレジスタの内容が交換される
```

b)xchg命令を使用

変数xとyの内容の交換を，xchg命令を用いて行う例を**リスト3．9**に示します．他のメモリやレジスタの内容を破壊することなく行うプログラム例です．8086の命令では，例外を除いて基本的にオペランドの両方をメモリアドレスにすることはできません．したがって，xchg x,yと記述することはできないわけです．そこでxchg命令を3度用いることによって，他のデータを破壊せずにデータ交換を行っています．

このプログラムでは，網掛け部分がxchg命令によるデータ交換に必要なデータとプログラム部分です．他は，行4:～5:がMS-DOS上で実行するために必要なセグメントレジスタの設定，行11:～17:がデータの表示のためのシステムコール，行19:～20:がMS-DOSへ復帰するためのシステムコールとなっています．

行7:のxchg x,alで，変数xとレジスタALの内容が交換されます．他のレジスタやメモリの内容が壊されることはありません．行8:のxchgでyとALの内容が交換されるので，結果的に，yにはxの内容が記入されたことになります．ここでALにはyの内容が記憶されています．さらに行9:のxchgでxとALの内容が交換されて，xには結果的にyが格納されたことになります．行9:のxchgでは，xに一時的に存在したALの初期データが，ALに記入され，元の状態に戻されます．

リスト3．9●xchg命令のプログラム例（xとyの文字を交換する）

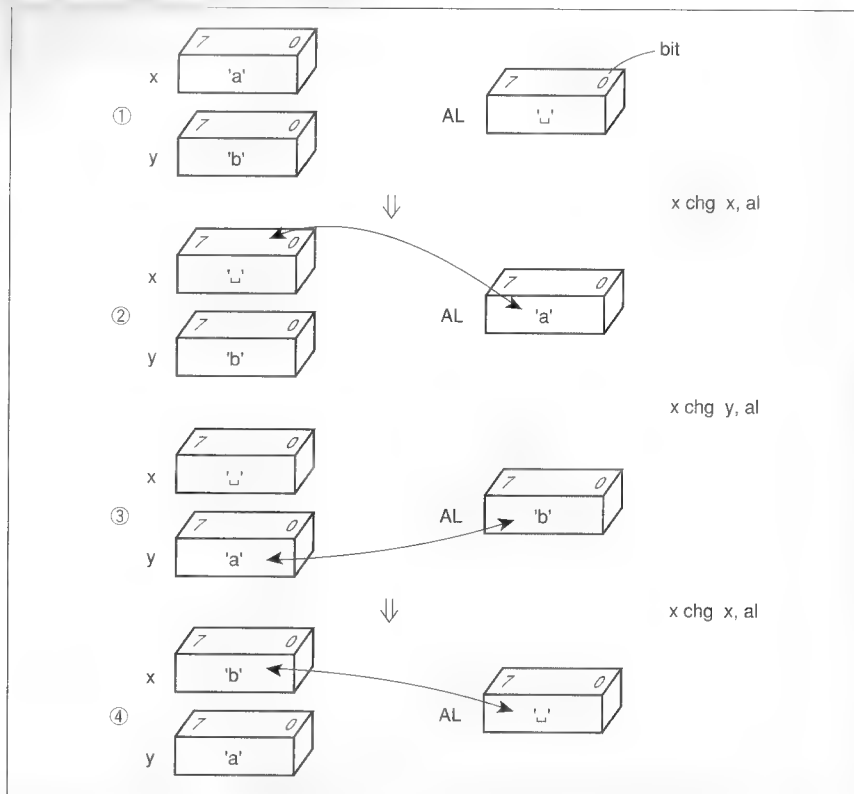
```
行  アドレス コード          プログラム              コメント

 1:                         .model small         ;メモリモデルsmall
 2:                         ;------------------------------------------
 3: 0000                    .code                ;コードセグメントの始まり
 4: 0000  B8 ---- R         start: mov ax,@data  ;データセグメントアドレス
 5: 0003  8E D8                    mov ds,ax     ;データセグメントを設定
 6:                         ;
 7: 0005  86 06 0000 R             xchg x,al     ;xとalの内容を交換
```

```
 8: 0009 86 06 0001 R          xchg y,al    ;yとalの内容を交換
 9: 000D 86 06 0000 R          xchg x,al    ;xとalの内容を交換
10:                      ;
11: 0011 8A 16 0000 R          mov dl,x     ;xのデータをdlレジスタへ移動
12: 0015 B4 02                 mov ah,2     ;MS-DOSファンクション2(文字表示)
13: 0017 CD 21                 int 21h      ;MS-DOSシステムコール
14:                      ;
15: 0019 8A 16 0001 R          mov dl,y     ;yのデータをdlレジスタへ移動
16: 001D B4 02                 mov ah,2     ;ファンクション2(文字表示)
17: 001F CD 21                 int 21h      ;MS-DOSシステムコール
18:                      ;
19: 0021 B8 4C00               mov ax,4c00H ;OS復帰  ファンクション4C
20: 0024 CD 21                 int 21h      ;MS-DOSシステムコール
21:                      ;-------------------------------
22: 0000                 .data              ;データセグメントの始まり
23: 0000 61              x     db  'a'      ;文字*で初期化する変数xを定義
24: 0001 62              y     db  'b'      ;初期化しない変数yを定義
25:                      ;-------------------------------
26:                            end start    ;startから実行開始
```

実行例

```
C:¥prg>xchg
ba
```

図3. 7にリスト3. 9によるデータの交換状況を示します．レジスタALには空白文字' ␣ 'が保存されているとします．図のように3回のxchg命令によって，xとyのデータが交換され，ALのデータが元の' ␣ 'に戻っていることがわかります．xchg命令を3回用いることによって，レジスタの内容を破壊することなく，変数xとyのデータ交換が可能になっています．

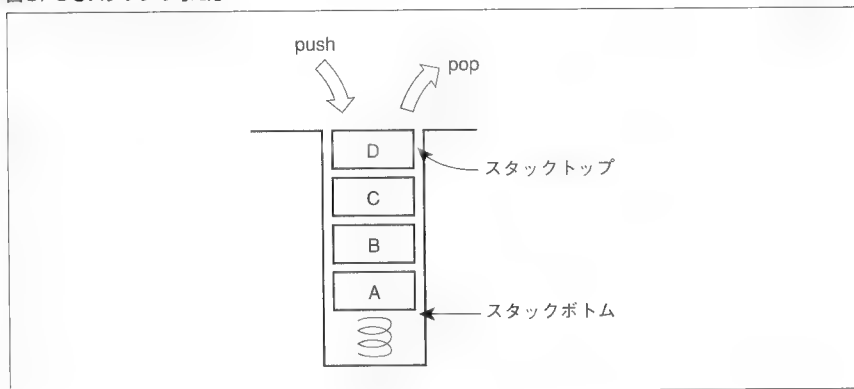
図3.7●xchgによるxとyのデータ交換

3.6 スタック命令

スタックとはメモリの一種で，メモリ番地を指定してアクセスするのではなく，データ保存した順とは逆順に取り出すことのできる(last in - first out)特殊なメモリです．一時的なデータの保存や，サブルーチンを呼び出す際のリターン番地の保存などに使われます．スタックの考え方を図3．8に示します．スタックとは，地面に穴を掘ってスプリングを底に取り付けたデータ保存場所だと考えてください．スタックにデータを保存することをプッシュ(pushまたはpush down)，スタックからデータを取り出すことをポップ(popまたはpop up)と呼びます．最初にデータAを保存し，次にB，C，Dの順にプッシュします．これを取り出そうとすると，最後にプッシュしたDしか取り出すことができません．プ

ッシュとポップの順は次のようになります．

pushの順　**A→B→C→D**
popの順　**D→C→B→A**

このように，保存と取り出しの順が逆になる特殊なメモリがスタックです．

図3．8●スタックの考え方

初期のマイクロプロセッサ8008などでは，特別にスタック機構が付加されていました．しかしスタックを特別に構成するため，8008のスタックは14bit×7個しか実現できず，実用面に問題がありました．そこで8086では，メモリ上に疑似的にスタックを構成し，スタックポインタSPを用いてその管理を行っています．

8086でのスタック実現方法を，push ax とpop axが実行された場合を例として**図3．9**に示します．スタックはスタックセグメント(64kバイト)としてメモリ上に構成されます．スタックトップを指すスタックポインタSP(stack pointer)が重要な働きをします．初期状態として図の①のように，SPをスタックボトムを指すように設定します．これはスタックが何もない状態であることを示しています．たとえばここでpush ax命令が実行されて，AXレジスタのデータがスタックへプッシュされたとすると，その動作は次のようになります．

```
push ax          ;(SP)←(SP)−2, ((SP):(SP)+1)←(AX)
```

ここで(SP)はSPの内容，((SP):(SP)＋1)はSPが指すメモリの内容(2バイト)を意味します．アドレスから−2をしたアドレスを指し，ここからワード単位でデータが保存されます．プッシュ後，SPは図上の②を指していることになります．SPは常に，最後にプッシュされたワード単位のデータ(スタックトップ)を指していることになります．スタックの生成方向は，普通のデータの生成方向と逆になっています．たとえば，データセグメントとスタックセグメントを同一の場所に設定した場合など，データ生成とスタック生成の方向が逆であることで，メモリを有効に使用できる場合があります．

ここでpop ax命令が実行されて，AXレジスタへデータがポップされるとすると，その動作は次のようになります．

```
pop ax          ;(AX)←((SP):(SP)+1),(SP)←(SP)+2
```

スタックトップからAXレジスタへワード単位でデータが回復されます．その後，SPの指すアドレスは，2増加することでポップしたデータは消されたことになります．

図3．9●スタックの実現

スタックに関する命令を**表3．4**に示します．スタック命令はレジスタ，変数の場合とフラグレジスタに関するものは異なります．レジスタや変数の場合はpush，pop命令のオペランドにワード単位でレジスタ，変数を指定します．フラグレジスタの場合は，pushf，popfのようして，オペランドは必要ないのです．

表3．4●スタック命令一覧

分類	命令	機能	
スタック	PUSH	push word onto stack	スタックへ保存
	POP	pop word off stack	スタックから復元
フラグ転送	PUSHF	push flags onto stack	フラグをスタックへ保存
	POPF	pop flags off stack	フラグをスタックから復元

スタック命令の使用法を，**リスト3．10**に示します．行12:にはデータセグメント，行15:ではスタックセグメントを定義してあります．.stack 100Hでは100Hバイトのスタック領域が作られます．

行2:～4:のpush命令では，レジスタまたは変数を指定して，スタックへデータを保存します．行4:のpush xでは，xに初期値として英字xyが与えられているため，スタックへはこの英字が保存されます．行5:のpush [di]では，DIレジスタ間接でメモリアドレスを指定しています．すなわち，DIが指すメモリの内容をスタックへ保存することになります．

行6:のpushfはオペランドがなく，フラグレジスタの内容をスタックへ保存し，行7:のpopfでは，スタックからフラグレジスタへ内容を回復しています．行8:のpop [si]では，行5:で保存したデータをSIが指すメモリ番地へ回復します．pushしたデータとpopしたデータの対応関係を，コメントの後ろに示してあります．これら対応関係は，誤りやすいものですから充分注意する必要があります．

リスト3．10●スタック命令

```
 1: .code
 2: push ax          ;AXの内容をプッシュ
 3: push si          ;SIの内容をプッシュ
 4: push x           ;変数xの内容をプッシュ
 5: push [di]        ;DIが指すメモリの内容をプッシュ
 6: pushf            ;フラグの内容をプッシュ
 7: popf             ;フラグへポップ
 8: pop [si]         ;SIが指すメモリアドレスへポップ
 9: pop y            ;変数yへポップ
10: pop cx           ;CXへポップ
11: pop bx           ;BXへポップ
12: .data            ;データセグメント
13: x   dw  'xy'     ;word変数　文字xyで初期化
14: y   dw  ?        ;word変数　初期化せず
15: .stack 100H      ;スタック領域100Hバイト
```


スタック命令を用いたプログラム例を**リスト3．11**に示します．変数wにある英字2文字をスタックへ保存し，これをレジスタDXへ回復して，確認のため

表示する簡単なプログラムです．

プログラムの網掛け部分がstack命令に必要なデータとプログラム部分です．他は，行3:～5:がMS-DOS上で実行するために必要なデータセグメントレジスタの設定，行11:～20:がデータの表示のためのMS-DOSシステムコール，行19:～20:がMS-DOSへ復帰するためのシステムコールとなっています．

行22:～23:がデータセグメントで，変数wをワードとして英字abで初期化しています．行25:の.stack 100Hでは，スタック領域を定義して100Hバイトの領域を設定しています．このプログラムでは，スタックの領域を定義するだけで，スタックポインタSPを初期化していません．これは，簡略化セグメント定義を用いた場合，スタックセグメントレジスタSSとスタックポインタSPを，アセンブラ側が自動的に設定してくれるからです．

完全なセグメント定義を行うプログラムでは，SSとSPの設定はプログラム上で明確に設定しなければなりません．スタックを用いた完全なセグメント定義プログラムについては，9章の『疑似命令とマクロ命令』と11章の『基本プログラミング』で説明しますので，必要な場合はそちらを参照してください．

実際のスタックに関連するのは**リスト3．11**の網掛け部分です．ここでは，網掛けの部分が理解できれば充分です．

行7:のpush wで，変数wの内容がスタックへ保存されます．ここでスタックに保存したデータをpop dxで，DXレジスタへ回復します．

回復できたデータをMS-DOSシステムコールで表示しているのが，行11:～17:の部分で，行19:～20:でMS-DOSへ戻ります．プログラムの実行例を，プログラムの後ろに示してあります．

リスト3．11●スタック命令プログラム（wの内容をプッシュ，ポップし，表示する）

```
行  アドレス コード            プログラム            コメント

 1:                        .model small
 2:                        ;-----------------------------------------
 3: 0000                   .code                 ;コードセグメント
 4: 0000  B8 ---- R        start: mov ax,@data   ;データセグメントのアドレス
 5: 0003  8E D8                   mov ds,ax     ;DSを設定
 6:                        ;------------------------------
 7: 0005  FF 36 0000 R            push w         ;変数wをスタックへ保存
```

```
 8: 0009 5A            pop dx        ;スタックからレジスタDXへ回復
 9:                ;-------------------------------
10:                ;スタックの内容を表示して確認する
11: 000A 86 D6         xchg dl,dh    ;DLとDHの内容を交換
12: 000C B4 02         mov ah,2      ;ファンクション番号2(1文字表示)
13: 000E CD 21         int 21h       ;MS-DOSシステムコール
14:                ;
15: 0010 86 D6         xchg dl,dh    ;DLとDHの内容を交換
16: 0012 B4 02         mov ah,2      ;ファンクション番号2(1文字表示)
17: 0014 CD 21         int 21h       ;MS-DOSシステムコール
18:                ;
19: 0016 B8 4C00       mov ax,4c00h ;ファンクション番号4C
20: 0019 CD 21         int 21h       ;MS-DOSシステムコール
21:                ;---------------------------------------------
22: 0000           .data             ;データセグメント
23: 0000 6162      w    dw  'ab'     ;変数w
24:                ;---------------------------------------------
25:                .stack 100H       ;スタックセグメント 100バイトの領域
26:                ;---------------------------------------------
27:                    end start     ;startから実行開始
```

実行例

```
C:¥prg>stck
ab
```

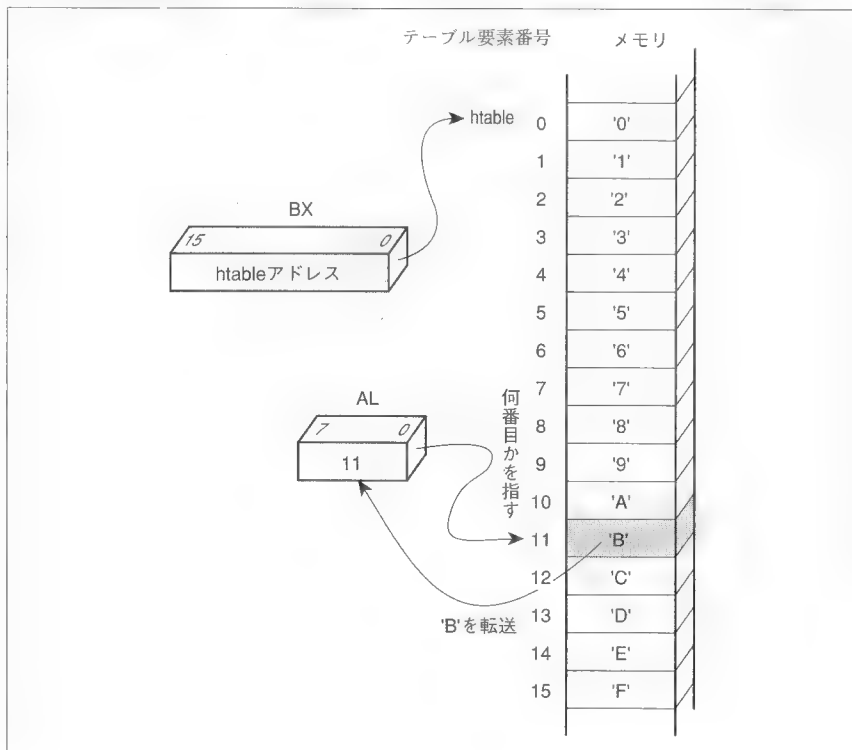
3.7 テーブルによる変換

8086にはテーブルを用いて数値を文字などに変換する便利な命令xlatがあります．xlat命令を用いて，0から15までの2進コードを16進数字に変換する例を取りあげます．まずメモリ上にテーブル(表)htableを作り，ここに16進数字'0'～'F'を格納します．xlat命令は**図3．10**に示すように，BXレジスタに表のアドレスを与えて，ALレジスタに変換しようとする2進数を与えます．たとえば，

```
lea   bx,htable
mov   al,11
xlat
```

のように用いると，レジスタALは11になり，xlat命令により結果として図のようにALに文字Bが転送されます．

リスト3．12に2進コードを16進数字に変換するプログラムを示します．行21:～22:には，2進コード00001011Bを初期値として与えた変数binがあり，16進数字のテーブルhtableがあります．htableには，0～Fまでの16進数字を初期値として与えています．実際の変換部分は行8:～10:で，lea bx,htableでテーブルのアドレスを与え，ALレジスタに変換する番号11を代入して，xlat命令でALレジスタに対応する文字Bが転送されます．実行例に結果として得られたBが表示されています．

図3．10●xlat命令

リスト3．12●テーブルによる変換（2進コードを16進数字に変換し，表示する）

```
行  アドレス コード          プログラム            コメント

 1:                          .model small
 2:                          ;
 3:                          ;------------------------------------------
 4: 0000                     .code                 ;コードセグメント
 5: 0000 B8 ---- R           start: mov ax,@data ;データセグメントアドレス
 6: 0003 8E D8                      mov ds,ax    ;データセグメントを設定
 7:                          ;
 8: 0005 8D 1E 0001 R               lea bx,htable ;16進テーブルの先頭番地を設定
 9: 0009 A0 0000 R                  mov al,bin   ;下位4bitを設定
10: 000C D7                         xlat    ;bxが指すテーブルの(al)番目の項目をalへ
11:                          ;
12: 000D 8A D0                      mov  dl,al   ;表示のためデータをdlへ
13: 000F B4 02                      mov ah,2     ;ファンクション2(文字表示)
14: 0011 CD 21                      int 21h      ;MS-DOSファンクションをコール
15:                          ;
16: 0013 B8 4C00                    mov ax,4c00H ;ファンクション4c
17: 0016 CD 21                      int 21h      ;MS-DOSシステムコール
18: ;------------------------------------------
19: 0000                     .data                 ;データセグメント
20:                          ;
21: 0000 0B                  bin    db  00001011B    ;2進数(10進 11)
22: 0001 30 31 32 33 34 35 htable db  '0123456789ABCDEF' ;16進文字テーブル
23: 36 37 38 39 41 42
24: 43 44 45 46
25:                          ;------------------------------------------
26: .stack 100H              ;スタックセグメント領域100H
27:                          ;------------------------------------------
28:                                 end start  ;プログラムの終了.start番地から実行


実行例
    C:¥prg>xlat
    B
```

演習問題3

1. 8086の絶対番地と相対番地はどのように表現されるか，簡単に説明せよ．

2. 次の命令を記述せよ．データはデータセグメント，コードはコードセグメントに分けること．
   (1) byte変数xの内容を，レジスタALへ転送する．
   (2) レジスタBHの内容を，byte変数yへ転送する．
   (3) byte変数xの内容を，byte変数yへ転送する．
   (4) word変数wの内容を，word変数vに転送する．
   (5) レジスタAXの内容を，byte変数hとlへ1byteずつ転送する．
   (6) word変数wの内容を，レジスタBHとBLへ転送する．
   (7) word変数aとword変数bの値を交換する．ただし，他のレジスタやメモリの内容を破壊しないこと．
   (8) セグメントレジスタDSに，1000Hを設定せよ．
   (9) バイト変数xの内容をyへ，yの内容をzへ，zの内容をxへ移動せよ．
   x→y→z→x
   (10) 変数xを間接アドレスを用いて，変数yへ転送せよ．
   (11) word変数aとbの内容をスタックへ保存し，これをAXレジスタとBXレジスタへ回復する．対応はa→AX，b→BXとする．

3. 次のプログラムを記述し，アセンブルして実行せよ．
   (1) strg番地にある文字列をディスプレイ表示する．文字列の最後には$があるものとする．
   (2) ディスプレイ上に，自分に関連すること（氏名，趣味など）を表示する．なるべくディスプレイ全面を使用し，画面構成を工夫して独自の美的感覚を示すこと．

第4章

算術演算命令

本章では，8086の算術演算命令について具体的に説明します．8086の演算は整数に対するものですが，例題では演算結果を表示することで，演算の正しさを確認しています．8086での演算結果は2進コードになっているため，これをディスプレイ表示するために，2進コード－10進数字変換のサブルーチンを用います．サブルーチンについては後章で説明しますから，単に表示するツールとして用いてください．

8086の算術演算命令は，8ビットデータと16ビットデータについて整数演算を可能にしています．8086の演算ブロックを**図4．1**に示します．汎用レジスタと表示されているレジスタが演算に用いることのできるレジスタです．演算命令を実行すると演算の結果が得られますが，演算結果が0である，演算結果が一(負)である，などのように演算結果の状態を示す各1ビットのデータもセットされます．この状態を示すデータはフラグと呼ばれ，フラグレジスタ(FLAGS)に保存されます．フラグレジスタは各1ビットのさまざまなフラグを集めたものです．演算結果の状態を示すフラグは重要で，後に述べるようにプログラムの流れを制御するのに用いられます．

図4．1●8086の演算ブロック

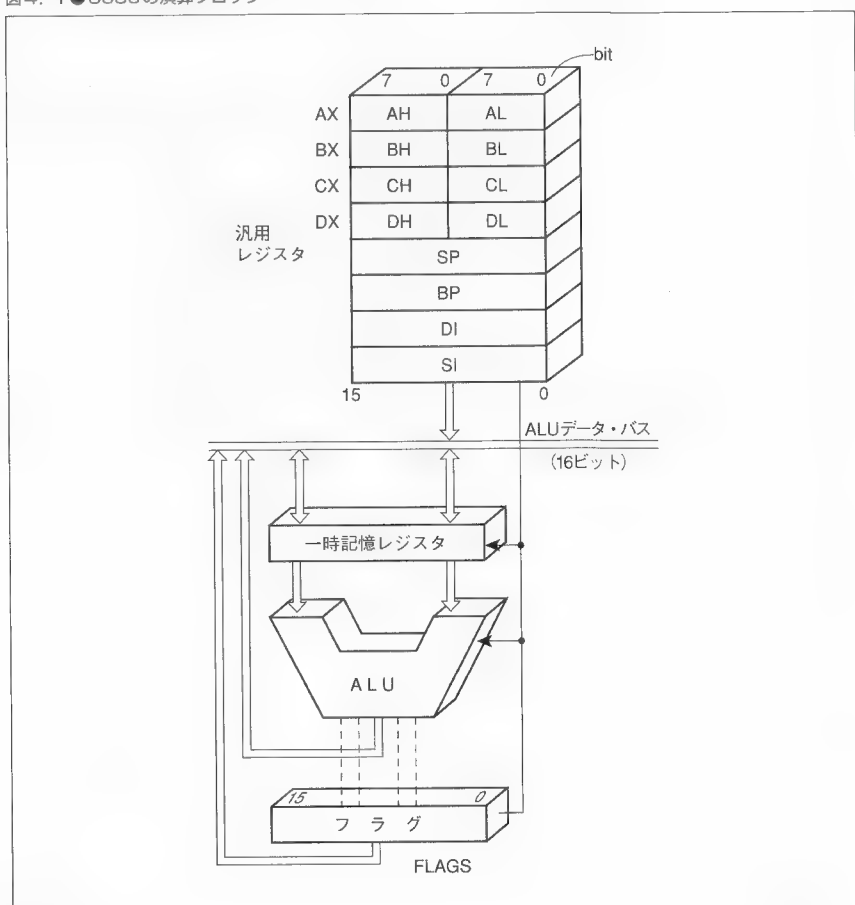



表4．1に8086の算術演算命令を示します．算術演算命令には整数の加減乗除，データの算術比較，データの増減，データ幅の拡張，ASCII演算の補正，10進演算の補正に関する命令などがあります．

表4．1●算術演算命令

	命令	機能	
加減算	ADD	add byte or word	加算
	ADC	add byte or word with carry	桁上げを含めた加算
	SUB	subtract byte or word	減算
	SBB	subtract byte or word with borrow	借りを含めた減算
増減	INC	increment byte or word	インクリメント
	DEC	decrement byte or word	ディクリメント
乗除算	MUL	multiply byte or word unsighned	符号のない乗算
	IMUL	integer multiply byte or word	符号付き乗算
	DIV	divide byte or word unsigned	符号のない除算
	IDIV	integer divide byte or word	符号付き除算
データ変換	CBW	convert byte to word	byteを符号付きでwordに変換
	CWD	convert word to byte	wordを符号付きでdwordに変換
データ比較	CMP	compare byte or word	減算をしてフラグをセット
ASCII演算補正	AAA	ASCII adjust for addition	ASCII加算後の補正
	AAS	ASCII adjust for subtraction	ASCII減算後の補正
	AAM	ASCII adjust for multiply	ASCII乗算後の補正
	AAD	ASCII adjust for division	ASCII除算前の補正
10進演算補正	DAA	decimal adjust for addition	10進加算後の補正
	DAS	decimal adjust for subtraction	10進減算後の補正

4.1 加減算命令

ここでは算術演算命令のうち加減算に関する命令をとりあげます．リスト4．1に加算命令の使用例を示します．データ転送命令で示したように，8086には多くのアドレシングモードがありますが，ここでは複雑なアドレシングは避けて，表3．3に示した単純化したアドレシングモードを用います．

行1:では.code簡略化疑似命令によって，命令がコードセグメントにあることを示しています．前述のように8086ではデータがどのセグメントにあるかで生成されるコードが異なってくるために，セグメントを示してあります．

行2:～7:のadd al,5では，ALレジスタの値と定数5を加算し，加算結果をALへ保存します．演算命令でもデータ転送命令と同様に

```
add     destination,source
```

のように，最初のオペランドがディスティネーションで，2番目のオペランドがソースとなります．データ転送命令と異なるのは，ディスティネーションの値も演算に用いられて，計算結果がディスティネーションに保存されることです．

ディスティネーションには，汎用レジスタのAX(AH,AL)，BX(BH,BL)，CX(CH,CL)，DX(DH,DL)，SP，BP，SI，DIおよび変数などのメモリアドレスを指定できます．またBX，BP，SI，DIレジスタでは間接的にメモリアドレスを指定できます．ソースには，ディスティネーションで指定できるレジスタとメモリアドレスだけでなく，定数を指定することもできます．

行3:のadd bx,512では16ビットのBXレジスタと定数512が加算され，結果がBXレジスタへ保存されます．add cl,bでは，CLレジスタと変数bの内容を加算し，結果をCLへ保存します．add w,dxではワード変数wとDXを加算して，結果をwへ保存します．add ax,bxではレジスタAXとBXの内容を加算し，AXへ保存します．

行7:〜8:は間接アドレスでの加算命令の例です．add al,[si]ではレジスタALと，SIが指すメモリ番地の内容を加算し，結果をALへ保存します．この場合，SIが指すメモリ番地の内容は，バイトデータでないと加算結果が意味のないものになります．add [di],axはDIレジスタが指すメモリ番地の内容とAXを加算し，結果をDIレジスタが指すメモリ番地へ保存します．

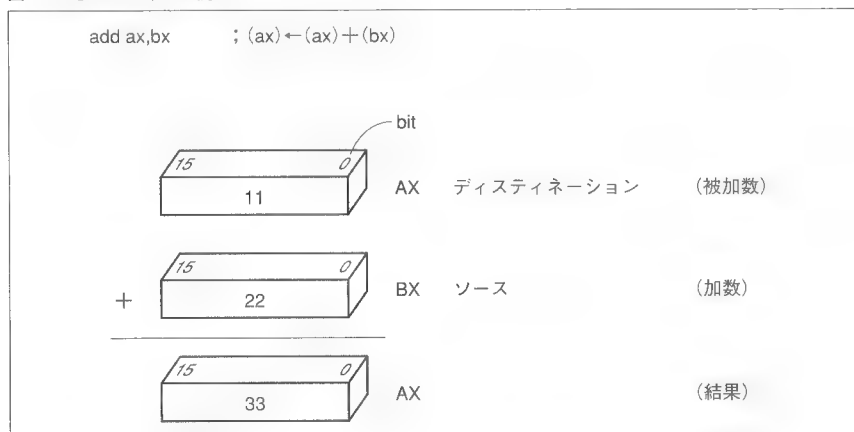
行9:の.dataはデータセグメントの始まりであることを示します．ここでbがバイト変数で初期値3が与えられることを意味します．wがワード変数で，初期値が1024であることを示します．

リスト4．1●加算命令

```
1: .code
2:      add al,5            ;alと直接データ5の加算
3:      add bx,512          ;bxと直接データ512との加算
4:      add cl,b            ;clと変数bとの加算
5:      add w,dx            ;変数wとdxとの加算
6:      add ax,bx           ;axとbxとの加算
7:      add al,[si]         ;alとSIが指すメモリの内容との加算
```

```
 8:     add [di],ax             ;DIが指すメモリの内容とAXとの加算
 9: .data
10: b   db  3
11: w   dw  1024
```

行6:の演算命令add ax,bxを例にした，ソースとディスティネーションの関係を図4．2に示します．AXレジスタには10進数の11が，BXレジスタには22があるものとします．ここでadd命令を実行すると，演算結果はAXに入り，前の値は保持されません．これに対してソースオペランドのBXの値は元のままで残ります．このソースとディスティネーションオペランドの関係は，2つのオペランドを持つ演算命令では同様になります．

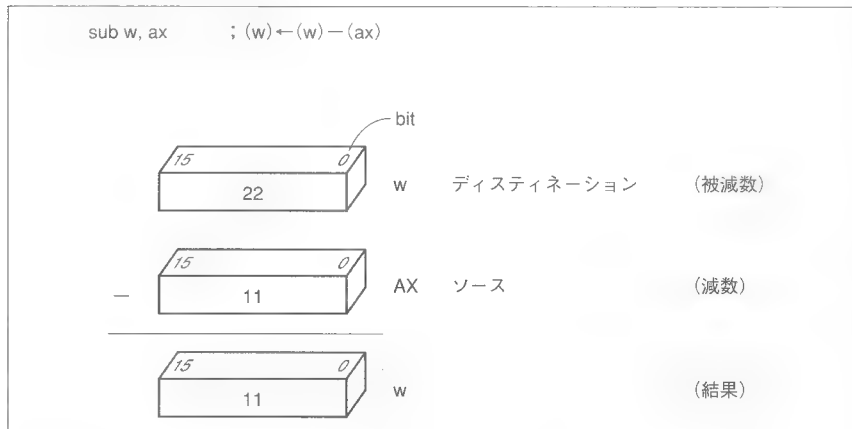
図4．2●add ax,bxの例

加算命令が実行されると，加算結果の他に，加算結果の状態を示すフラグがセットされます．加算命令で状態がセットされるのは，A(補助キャリ)，C(キャリ)，P(パリティ)，S(サイン)，Z(ゼロ)，O(オーバフロー)フラグの6つのフラグです．これらフラグは，条件付きジャンプ命令の条件として用いることができます．

リスト4．2に減算命令の使用例を示します．ディスティネーション－ソースの減算結果をディスティネーションに保存することを除けば，加算の場合と同

様となります．減算命令sub w,axにおけるソースとディスティネーションの関係を図4．3に示します．変数wには10進数の22が，AXレジスタには11があるものとします．ここでsub命令を実行すると，演算結果は変数wに保存されます．これに対してソースオペランドのAXの値は元のままで残ります．

リスト4．2●減算命令

```
 1: .code
 2:      sub dl,3          ;DLから直接データ3を引く
 3:      sub cx,100        ;CXから直接データ100を引く
 4:      sub bl,b          ;BLから変数bの値を引く
 5:      sub w,ax          ;変数wからAXの値を引く
 6:      sub cx,dx         ;CXからDXの値を引く
 7:      sub bl,[si]       ;BLからSIが指すメモリの内容を引く
 8:      sub [di],ax       ;DIが指すメモリの内容からAXの値を引く
 9: .data
10: b    db   3
11: w    dw   512
```

図4．3●sub w,axの例

加減算命令を用いたプログラム例を，リスト4．3に示します．変数xから変数yを減算し，答を変数zへ保存した後に，MS-DOSシステムコールを用いて表示する例です．

データは行18:～20:で，変数xに5が，変数yに3が初期値として与えられて

います．行7:のmov al,xでは，変数xの値をALレジスタへロードします．8086命令では，基本的にメモリーメモリ間の移動や演算はできません．そこでxの値をALに保存し，yとの減算を行います．結果はALレジスタに得られるので，これを変数zへ保存するのが，mov z,al命令です．

この後，計算結果を表示します．ALにある計算結果は，2進数で表現されています．これをそのままMS-DOSシステムコールの文字出力に与えても，結果が正しく表示されません．コンピュータ内部では，数値と文字の扱いが異なっているために変換が必要になります．一般に数値は純2進数で表現され，アルファベットや数字などは，ASCIIコードで表現されます．付録にASCIIコード表を付けてあるので参考にしてください．

0～9までの数値なら，ASCIIコードがわからなくても表示する方法があります．付録のASCIIコード表を見ると，数字'0'は30_{16}(48_{10})，1は31_{16}(49_{10})のように順に並んでいることがわかります．もし数字'0'に純2進数の2を加算したらどうなるかを，**図4．4**に示します．結果は図に示すように，ASCIIコードの数字'2'になります．0～9までの純2進数なら，その値にASCIIコードの'0'を加えてやれば直ちに数字にすることができます．

これを用いて純2進数を文字に変換しているのが行9:の

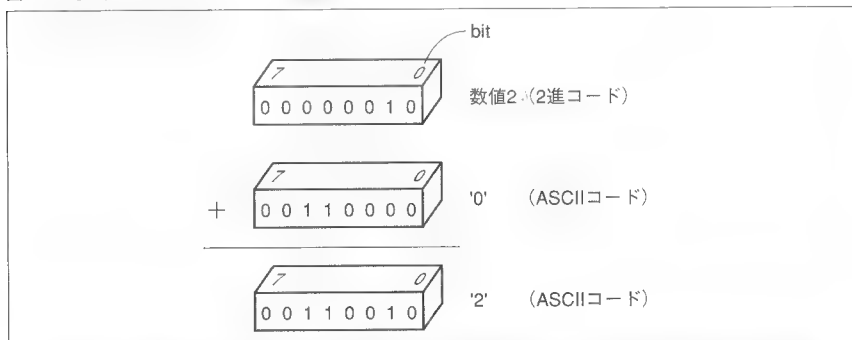
```
add al,'0'
```

です．これによって0～9までの数なら，直ちに数字に変換することができます．実行結果がプログラムの下に示してあります．

リスト4．3●加減算命令のプログラム例（x-yの結果を表示する）

```
行  アドレス コード         プログラム              コメント

 1:                        .model small          ;メモリモデルsmall
 2:                        ;-------------------------------------------
 3: 0000                   .code                 ;コードセグメントの始まり
 4: 0000  B8 ---- R        start: mov ax,@data   ;データセグメントアドレス
 5: 0003  8E D8                   mov ds,ax     ;データセグメントを設定
 6:                                              ;
 7: 0005  A0 0000 R                 mov al,x     ;xのデータをalへ移動
```

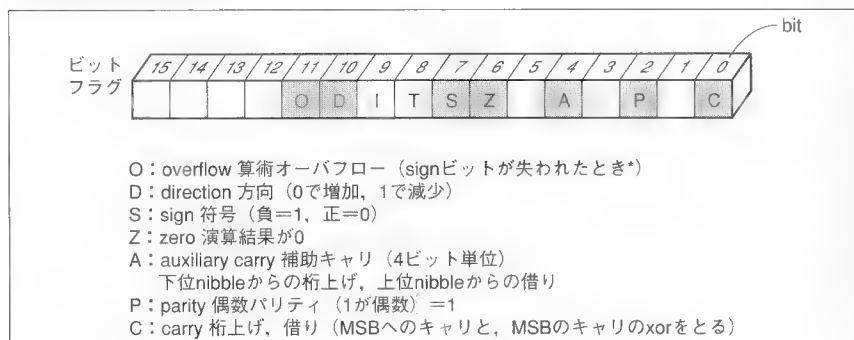
```
 8: 0008 2A 06 0001 R          sub al,y       ;(al)←(al)-(y)
 9: 000C 04 30                 add al,'0'     ;数字にするため、文字0を加算
10:                                           ;
11: 000E 8A D0                 mov dl,al      ;alのデータをdlレジスタへ移動
12: 0010 B4 02                 mov ah,2       ;ファンクション2(文字表示)
13: 0012 CD 21                 int 21h        ;MS-DOSシステムコール
14:                                           ;
15: 0014 B8 4C00               mov ax,4c00H ;OS復帰　ファンクション4C
16: 0017 CD 21                 int 21h        ;MS-DOSシステムコール
17:                     ;--------------------------------
18: 0000                .data                 ;データセグメントの始まり
19: 0000 05             x      db   5         ;変数xを初期値5で定義
20: 0001 03             y      db   3         ;変数yを初期値3で定義
21:                     ;--------------------------------
22:                            end start      ;startから実行開始

実行例
    C:¥prg>a&s
    2
```

図4．4●2進コード→ASCIIコード変換

4.2 フラグと比較命令

コンピュータ上で加算や減算が行われると，演算結果と演算の状態を示すフラグ(flag)がセットされます．フラグは「演算の結果が0となった」，「オーバフローが生じた」というような状態を，1ビットのデータとして表現するものです．

8086ではフラグレジスタが各種のフラグを保持しています．演算に関連する

フラグとしては，O(オーバフロー：overflow)，S(サイン：sign)，Z(ゼロ：zero)，A(補助キャリ：auxiliary carry)，P(パリティ：parity)，C(キャリ：carry)フラグがあります．これらのフラグは演算の結果によって，それぞれセット(1)，またはクリア(0)されます．またストリング命令に関してD(方向：direction)があり，割り込みに関連してI(割り込み：interrupt)，T(トラップ：trap)フラグがあります．今後表記上の混乱を避けるため，各フラグ名の後ろにFを付けて表記します(たとえば，AF)．

図4．5●フラグレジスタ

オーバフローフラグ(OF)は，算術オーバフローが生じたときに1にセットされます．これはデータの符号(sign)ビットが失われたときを意味します．

サインフラグ(SF)は，演算結果が正か負かを示します．すなわち符号ビット(MSB)が1のときは負を意味し，SFは1になります．符号ビットが0のときは正なので，SFは0となります．

ゼロフラグ(ZF)は，演算結果が0のとき1に，演算結果が0でないとき0になります．演算結果が0のときにゼロフラグ(ZF)が1になるというのは，感覚的には混乱を招くかも知れませんが，1は旗があがった状態だと考えればわかりやすいでしょう．

補助キャリフラグ(AF)は，BCDデータなど4ビット単位のデータの加減算で，下位4ビットからの桁上げを生じた場合か，上位4ビットからの借りが生じた場合に1にセットされます．このようなフラグがなぜあるのか不思議に思われるかも知れません．AFフラグは10進演算を行う場合に有効なフラグです．10進数

は一般にコンピュータ内部ではBCDコードで表現されます．BCDコードは，4ビットで数値の0～9までを表現します．BCDコードを用いて演算を行った場合，4ビットの幅を超えて桁上げが生ずる場合があります．たとえば8＋9の演算を行うような場合です．このような場合はBCD1桁が表現できる範囲を超えた値となり，何らかの補正を行う必要があります．このような場合，このAFを用いることができます．

パリティフラグ(PF)は，演算の結果，データ中の1の数が偶数であるとき1に，奇数であるとき0になります．すなわち偶数パリティであるかどうかを示しています．

キャリフラグ(CF)は，演算の結果，桁上げや借りが生じた場合に1になり，これらが生じなければ0になります．CFは実際上，最上位ビットへのキャリと，最上位からのキャリのxorをとって設定されます．

ディレクションフラグ(DF)は，ストリング命令で文字列の処理をメモリアドレスの大きい方向に進めるか，小さい方向に進めるかを決めるフラグです．この方向はSI，DIレジスタの自動増減の方向を決めることになり，0では増加方向，1では減少方向となります．

フラグは演算によっては，影響を受けない場合があります．どのフラグがどのようにセット／クリアされるかは，付録の8086命令一覧に示してあります．

フラグは直接その値を見るのではなく，フラグ関連の命令によって間接的に参照することが多いのです．しかし，あらかじめフラグをセット／クリアしておかなければならない場合があります．フラグを操作する命令を表4．2に示します．キャリフラグ(CF)，方向フラグ(DF)と割り込みフラグ(IF)の操作命令が用意されています．

表4．2●フラグ操作命令

分類	命令	機能	
フラグ操作	STC	set carry flag	キャリフラグをセット
	CLC	clear carry flag	キャリフラグをクリア
	CMC	complement carry flag	キャリフラグを反転
	STD	set direction flag	方向フラグをセット
	CLD	clear direction flag	方向フラグをリセット
	STI	set interrupt enable flag	割り込み許可フラグをセット
	CLI	clear interrupt enable flag	割り込み許可フラグをクリア

リスト4．4にフラグ操作命令の使用例を示します．フラグ操作命令にはオペランドはなく，キャリフラグに関するstc(CFをセット)，clc(CFをクリア)，cmc(CFを反転)命令が実行されると，フラグレジスタのCF(bit0)だけが影響を受けて，他のフラグは影響を受けません．同様にディレクションフラグに関するstd(DFをセット)，cld(DFをクリア)命令の実行では，DF(bit8)だけが影響を受けます．

リスト4．4●フラグ操作命令の使用例

```
1: .code
2:     stc                      ;キャリフラグをセット
3:     clc                      ;キャリフラグをクリア
4:     cmc                      ;キャリフラグを反転
5:     std                      ;方向フラグをセット
6:     cld                      ;方向フラグをクリア
```

フラグは加減算などの演算によってセットされますが，2つのデータを比較することでフラグを立てる命令があります．比較命令cmpは2つのデータを比べるための命令で，フラグをセットするために用います．

```
cmp destination,source
```

の形となります．

リスト4．5に比較命令の使用例を示します．cmp命令は基本的にsub命令と変わりありません．destination－sourceの計算をして，フラグをセットします．ただし，減算結果がディスティネーションに保存されることはありません．単にフラグをセットするだけの命令です．フラグを除いて他のレジスタやメモリには影響を与えないのです．cmp命令は後に出てくる条件付きジャンプ命令と組み合わせることで意味を持ちます．

リスト4．5●比較命令の使用例

```
 1: .code
 2:     cmp dl,7          ;DLと直接データ7を比較
 3:     cmp ax,256        ;AXと直接データ256を比較
 4:     cmp bl,b          ;BLと変数bを比較
 5:     cmp w,cx          ;変数wとCXを比較
 6:     cmp dx,ax         ;DXとAXを比較
 7:     cmp bl,[si]       ;BLとSIが指すメモリの内容を比較
 8:     cmp [di],cx       ;DIが指すメモリの内容とCXを比較
 9: .data
10: b   db  127
11: w   dw  1000
```

図4．6に比較命令を実行したときのフラグレジスタの状態を示します．ALレジスタに数値3があるときに，cmp al,5 を実行した場合です．影響を受けるフラグは網掛けしてあります．3－5の演算の結果，当然オーバフローは生じないので，OF＝0となります．保存されてはいませんが，演算結果は－2となるためサインフラグSF＝1となります．3－5の演算を行うとき，被減数3より減数5が大きいため，演算上では借りが生じます．この状態を示すため，キャリフラグCF＝1となります．同じく下位4ビット(nibble)で借りが生じたため，補助キャリフラグAF＝1となります．減算結果を2進数で示すと11111110_2となります．このとき，演算結果で1が立っているビットの数は7で，奇数ビットに1が立っていることになります．8086でのパリティフラグPFは，演算結果の1の数が偶数個のときに1にセットされます．このため，3－5の計算結果ではPF＝0となります．

図4．6●比較命令とフラグレジスタ

なお，どのような命令でフラグがセットされるかは，付録の命令一覧で確認ができます．

4.3 演算幅の拡張

8086の数値データは8ビットのバイトと，16ビットのワード(2バイト)の整数データが基本となります．しかし，実際には4バイトデータが必要になることがありますし，さらにデータ幅の大きいデータが必要になることがあります．このような場合のために，add命令の他にadc(add with carry)命令があります．add命令とadc命令を組み合わせることで，データ幅の大きなデータの加算も可能になります．AXレジスタとBXレジスタを加算し，結果をAXレジスタへ保存するのは次の命令となります．

```
add ax,bx
```

これを，8ビットのレジスタAH，AL，BH，BLを用いて1バイトの演算として実現することができます．すなわち，データの下位8ビットをALとBLの加

算として，上位8ビットをAHとBHの加算として実現し，結果をAXレジスタに残すことができます．

これを下に示します．

①add al,bl

②adc ah,bh

①の加算では単にALとBLを加算しているだけですが，このとき加算結果が8ビットの範囲を超えて桁上げが生じてしまう場合があります．加算の結果，桁上げが生ずると，キャリフラグCF＝1となって，その状態を保存します．②のadc命令では，AHとBHを加算するだけではなく，キャリフラグを含めて加算が行われます．結果はAX(AH：AL)レジスタに，16ビットデータとして残されます．

図4．7にaddとadcを用いた16ビット加算の例を示します．AXレジスタには511が，BXレジスタには257があるものとします．AXレジスタをAHとAL，BXレジスタをBHとBLレジスタに分割し，AHとBH，ALとBLの加算を行います．加算は①，②の順に行います．①のadd al,blで加算結果は0で，生じた桁上げはキャリフラグCFへセットされます．このCFを含めた加算が②のadc ah,bhで行われます．例ではCFがセットされているため，AH,BH,CFの加算結果は3となります．AHレジスタの3はAXレジスタでは上位8ビットを示すことになるので，3×256＝768となります．これにALレジスタの0を加えると768となり，511＋257の加算結果が得られます．

図4．7●演算幅の拡張　加算

この方法はさらにワードデータ同士の加算によって，ダブルワードデータの加算に適用することもできます．たとえば，AXとBXで4バイトのデータを，CXとDXで他方の4バイトデータを保持するとすれば，次の命令によって4バイトデータの加算が実現できます．これを下に示します．

①add bx,dx

②adc ax,cx

結果は，AXに上位2バイトが，BXに下位2バイトが保存されることになります．

この結果をメモリ上に保存して，さらにadc命令を用いていけばデータ幅をメモリの許す限り拡張することができます．

この手法は，減算にも適用することができます．sub命令とsbb(subtruct with borrow)命令を用いて，バイト単位でワード減算を行う例を図4．8に示します．AXレジスタには33280が，BXレジスタには513があるものとします．AXレジスタをAHとAL，BXレジスタをBHとBLレジスタに分割し，AHとBH，ALとBLの減算を行います．減算は①，②の順に行います．①のsub al,blで減算結果は255で，生じた借りはキャリフラグCFへセットされます．このCFを含めた減算が②のsbb ah,bhで行われます．例ではCFがセットされているため，AH,BH,CFの減算結果は127となります．AHレジスタの3はAXレジスタでは上位8ビットを示すことになるので，127×256＝32512となります．これにALレジスタの255を加えると32767の減算結果が得られます．

図4．8●演算幅の拡張　減算

```
sub命令とsbb命令              8bitの減算2回で，16bitの減算をしたことになる

② sbb ah,bh                   ① sub al,bl

    1000 0010   AH              0000 0000 AL              33280
    0000 0010   BH              0000 0001 BL                513
 ー           1  CF          ー                         ー
 ─────────────────          ────────────────          ──────────
  1 0111 1111 ah             1 1111 1111 AL              32767
  CF                        借りCF

(ah)←(ah)ー(bh)ー(c)          (al)←(al)ー(bl)
```

4.4 インクリメント/ディクリメント命令

レジスタや変数の内容を1増加させるinc命令と，1減少させるdec命令の使用例をリスト4．7に示します．行2:～9:までのinc，dec命令ではレジスタと変数の内容を増減しています．行10:～11:ではBXレジスタを用いて，間接アドレスで変数の内容を増減しています．mov bx,offset xでBXレジスタに変数xのアドレスを設定します．dec byte ptr [bx]でxのアドレスを間接指定して，その内容を1減しています．ここでbyte ptrはBXレジスタが示すデータが，バイトであることをアセンブラに指示するために付けます．

リスト4．7●inc，dec命令の使用例

```
 1: .code                          ;コードセグメント
 2:     inc ax                     ;AXを1増
 3:     inc bl                     ;BLを1増
 4:     inc x                      ;変数xの内容を1増
 5:     inc y                      ;変数yの内容を1増
 6:     dec cx                     ;CXを1減
 7:     dec dh                     ;DHを1減
 8:     dec x                      ;変数xの内容を1減
 9:     dec y                      ;変数yの内容を1減
10:     mov bx,offset x            ;変数xのアドレスをBXへロード
11:     dec byte ptr [bx]          ;メモリの内容をBXで間接指定して1減
12: .data                          ;データセグメント
13: x    db 0                      ;変数x  0で初期化
14: y    dw 10                     ;変数y  10で初期化
```

プログラムではレジスタやメモリの内容を1増やしたり，1減らしたりすることが必要になることがよくあります．たとえば10回同じことを繰り返したいとき，CXレジスタに0を書き込んでおいて，1回処理するたびにCXを1ずつ増やして，CXの値が10になったなら処理を終了する場合などです．このような1ずつ増やす命令は，

```
add cx,1
```

と書くこともできますが，インクリメント命令を用いれば

```
    inc cx
```

のように，もっと簡単に処理できます．インクリメント命令は，数を数えるカウンタを用いるときに有効な命令です．

回数を数えるには，あらかじめ処理する回数を決めておいて，処理ごとに1ずつ減らしていき，0になったら終了するやり方もあります．この方法に用いるのがディクリメント命令です．

たとえば10回同じことを繰り返したいとき，CXレジスタに10を書き込んでおいて，1回処理するたびにCXを1ずつ減らしていき，CXの値が0になったなら処理を終了することができます．次のような命令によって実現できます．

```
    mov cx,10
lp: dec cx
    jnz lp
```

ここでjnz lpは条件付きジャンプ命令で，dec cx命令で，CXの値が0でないならラベルlpに戻って繰り返すことになります．条件付きジャンプ命令については，後の章で詳しく説明します．

リスト4．8に，変数xの値を繰り返しインクリメントして，最終の値を表示する簡単なプログラム例を示します．行23:で変数xに初期値0を与えています．実際の処理部分は行7:～10:の部分になります．mov cl,5で，カウンタとして使用するCLレジスタに5を与えています．inc xで変数xの内容をインクリメントし，dec clでカウンタCLをディクリメントします．jnz lpでディクリメントしたCLの値が0でなければ，lpへジャンプして行8:～10:の命令を繰り返します．CLの初期値は5でしたから，ループが終了した時点ではxの値は5になっているはずです．実行例でxの値は，当然5と表示されています．

リスト4．8●inc，dec命令のプログラム例（xをCL回数インクリメントし，表示する）

```
行   アドレス コード          プログラム              コメント

 1:                          .model small          ;メモリモデルsmall
 2:                          ;---------------------------------------------
 3: 0000                     .code                 ;コードセグメントの始まり
 4: 0000 B8 ---- R           start: mov ax,@data   ;データセグメントアドレス
 5: 0003 8E D8                      mov ds,ax      ;データセグメントを設定
 6:                          ;
 7: 0005 B1 05                      mov cl,5       ;(cl)←5
 8: 0007 FE 06 0000 R        lp:    inc x          ;(x)←(x)+1
 9: 000B FE C9                      dec cl         ;(cl)←(cl)-1
10: 000D 75 F8                      jnz lp      ;decの結果が0でないなら、lpへジャンプ
11:                          ;
12: 000F A0 0000 R                  mov al,x       ;(al)←(x)
13: 0012 04 30                      add al,'0'     ;数字にするため、文字0を加算
14:                          ;
15: 0014 8A D0                      mov dl,al      ;alのデータをdlレジスタへ移動
16: 0016 B4 02                      mov ah,2       ;ファンクション2(文字表示)
17: 0018 CD 21                      int 21h        ;MS-DOSシステムコール
18:                          ;
19: 001A B8 4C00                    mov ax,4c00H   ;OS復帰　ファンクション4C
20: 001D CD 21                      int 21h        ;MS-DOSシステムコール
21:                          ;-------------------------------
22: 0000                     .data                 ;データセグメントの始まり
23: 0000 00                  x      db  0          ;変数xを初期値0
24:                          ;-------------------------------
25:                                 end start      ;startから実行開始


実行例
    C:¥prg>inc
    5
```

4.5 乗除算命令

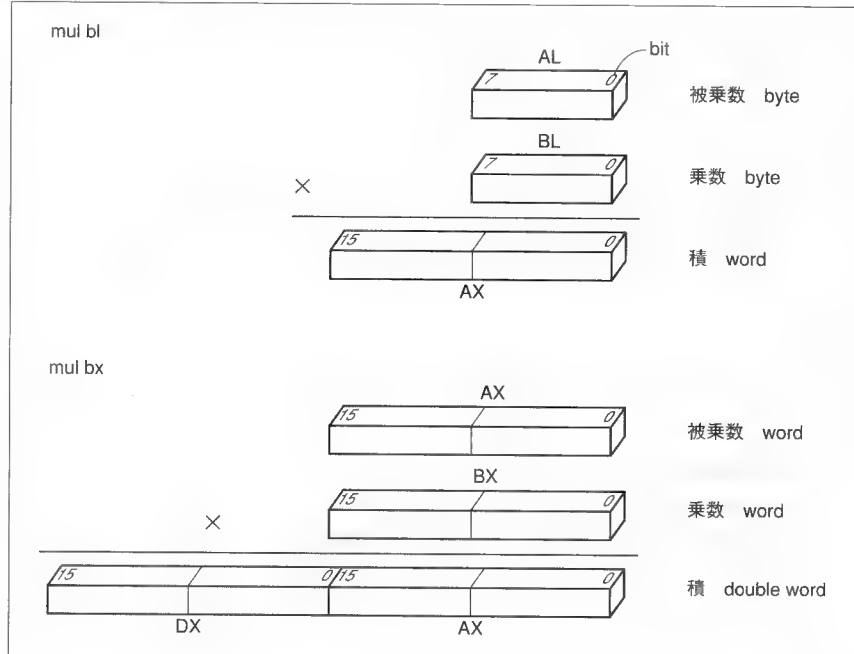
8086で持っている乗除算は，加減算と同じく整数の計算です．乗除算では符号の付いたデータでも，付かないデータでも演算可能になっています．またデータはバイトとワードのいずれでも可能です．

乗算では，演算後のデータは倍の幅に拡張されます．バイトデータの乗算結

果はワードに，ワードデータの乗算結果はダブルワードに拡張されます．符号のない数値の乗算は次の形となります．結果もまた符号のないデータとなります．

```
mul bl        ;(AX)←(AL)×(BL)     積←被乗数×乗数
mul bx        ;(DX:AX)←(AX)×(BX)
```

オペランドは1つで被乗数×乗数のうち乗数だけを示し，レジスタ，メモリ，間接メモリが許されます．イミディエート(直接データ)は許されないので注意が必要です．被乗数は暗黙のうちにALまたはAXにあるものとされ，オペランドには示されません．オペランドがバイトの場合はALに，ワードの場合はAXに被乗数があるものとされます．mul blでは，レジスタALとBLのバイトの乗算結果はAXにあり，ワードに拡張されています．mul bxでは，レジスタAXとBXの乗算結果は，DXとAXを結合したダブルワードに拡張されます．この様子を図4．9に示します．

図4．9●乗算命令

符号付きの乗算には，imul命令を用います．データを符号付きと見なして演算する他はmul命令と同じ働きとなります．

```
imul bl        ;(AX)←(AL)×(BL) 積←被乗数×乗数
imul bx        ;(DX:AX)←(AX)×(BX)
```

mul命令のプログラム例を**リスト4．9**に示します．変数xとyの乗算結果をzに保存し，確認のために16進表示を行います．MS-DOSシステムコールはマクロ化したインクルッドファイルiomac.incを読み込んで用いることにします．行3:のincludeによって，アセンブル時にiomac.incファイルが読み込まれます．16進表示のためには，サブルーチンhexWを呼び出します．サブルーチンは一連の命令を実行する独立したプログラムで，通常機能ごとに作ります．行4:のextrn hexWで，hexWがこのプログラム内にはなく，他のファイルにあるサブルーチンであることを示しています．サブルーチン(手続き)hexWは別ファイルに保存されており，このプログラムと同時にリンクして実行形式のプログラムを作ります．

行3:でinclude iomac.incでは，MS-DOSシステムコールをマクロ定義したファイルを読み込みます．このプログラムでは行17:のexitがマクロ定義の使用方法となります．定義した名前を書くことで，使用可能になります．

実際の処理部分は行11:～13:までの命令で，被乗数xをALレジスタにロードし，これと被乗数yの乗算結果AXをzに保存します．行15:のcall hexWでサブルーチンを呼び出して，乗算結果を16進表示します．5×3の乗算結果が実行例に示してあります．

リスト4．9●mul命令のプログラム（x×yの結果を表示する）

```
1: .model small              ;メモリモデルsmall
2: ;---------------------------
3: include    iomac.inc      ;MS-DOSシステムコールのマクロ定義ファイル
4: extrn hexW :near          ;axの内容を16進表示する外部手続き
5: ;-------------------------------------
6: .code                     ;コードセグメントの始まり
7: ;
```

```
 8: start:      mov ax,@data  ;データセグメントアドレス
 9:             mov ds,ax     ;データセグメントを設定
10: ;
11:             mov al,x      ;xのデータをalへ移動
12:             mul y         ;(ah:al)←(al)*(y)
13:             mov z,ax      ;乗算結果をzへ保存
14: ;
15:             call hexW     ;16進表示手続きの呼びだし.AXにデータ
16: ;
17:             exit          ;MS-DOS復帰(マクロ)
18: ;------------------------
19: .data                     ;データセグメントの始まり
20: x           db 5          ;変数xを初期値5で定義
21: y           db 3          ;変数yを初期値3で定義
22: z           dw ?          ;ワード変数zを初期化せずに定義
23: ;--------------------------------------
24: .stack      100H          ;スタックセグメント
25: ;--------------------------------------
26:             end start     ;startから実行開始
```

実行例

```
C:¥prg>mul
000F
```

ここで，16進表示サブルーチンhexW(numlib.asmファイル)，入出力マクロ定義disp(iomac.asmファイル)などの利用法を簡単に説明します．この時点ではプログラムの内容ではなく，その利用法を理解してもらえれば充分です．これらは以下のように，別ファイルにあるものを利用します．

サブルーチンhexW,decW,binWなど	numlib.asm	第6章5節で定義
マクロkeyin,disp,exit,crlfなど	nummac.inc	第10章3節で定義

サブルーチンhexWは，AXレジスタにある2進数を16進4桁で表示します．これを用いるプログラム内に，次のように記述することで利用できます．

```
extrn hexW          ;外部手続きであることの宣言
mov ax,z            ;AXに2進数を代入．変数zにあるとする．
call hexW           ;実際の手続き呼び出し
```

サブルーチンhexWを含んだファイルnumlib.asmは，メインプログラムとは別にアセンブルし，下のようにメインプログラムとリンクして1つのプログラムにする必要があります．

```
C:¥prg>link mul numlib    mul.objとnumlib.objのリンク → mul.exe
```

マクロ命令dispは，MS-DOSシステムコールでキーボード入力を行います．これはファイルiomac.asmで定義されており，次のようなマクロ命令が含まれています．

```
keyin       ;キーボードから1文字入力する
disp        ;ディスプレイに1文字表示する
exit        ;MS-DOSへ復帰する
crlf        ;表示を1行改行する
```

マクロ命令を引用する場合，

```
exit
```

という名前を記述すだけで，マクロ定義で記述された命令群がその場所に展開されます．マクロ定義については，第9章『疑似命令とマクロ命令』を参照してください．

除算では，データの幅を倍にしてから演算を行い，結果は元のデータ幅となります．バイトデータはワードに，ワードデータはダブルワードに拡張してから除算を行い，それぞれ元のデータ幅の商と余りが得られます．

符号のない数値の乗除算は次の形となります．結果もまた符号のないデータとなります．

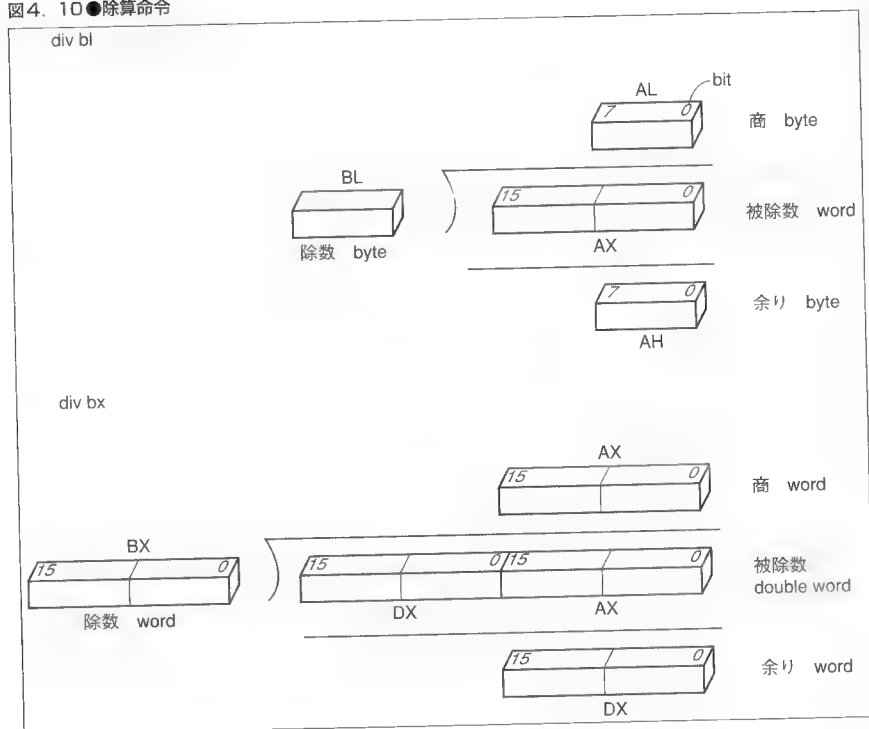
```
div bl      ;(AL)←(AX)/(BL)         商←被除数／除数
            ;(AH)←(AX)%(BL)         余り
div bx      ;(AX)←(DX:AX)/(BX)      商　被除数／除数
            ;(DX)←(DX:AX)%(BX)      余り
```

オペランドは1つで除数を示し，レジスタ，メモリ，間接メモリが許され，イミディエート(直接データ)が許されないのは乗算と同様です．被除数は暗黙のうちにAXまたはDX：AX(DXとAXを結合した2ワード)にあるものとされ，オペランドには示しません．オペランドに示すのは除数で，これがバイトの場合AXに被除数が，ワードの場合はDX:AXに被除数があるものとされます．

div blでは，レジスタAXとBLの除算結果の商はALに，余りがAHに保存されます．div bxでは，レジスタDX:AXとBXの除算結果はAXへ，余りがDXへ保存されます．

```
idiv bl     ;(AL)←(AX)/(BL)         商←被除数／除数
            ;(AH)←(AX)%(BL)         余り
idiv bx     ;(AX)←(DX:AX)/(BX)      商←被除数／除数
            ;(DX)←(DX:AX)%(BX)      余り
```

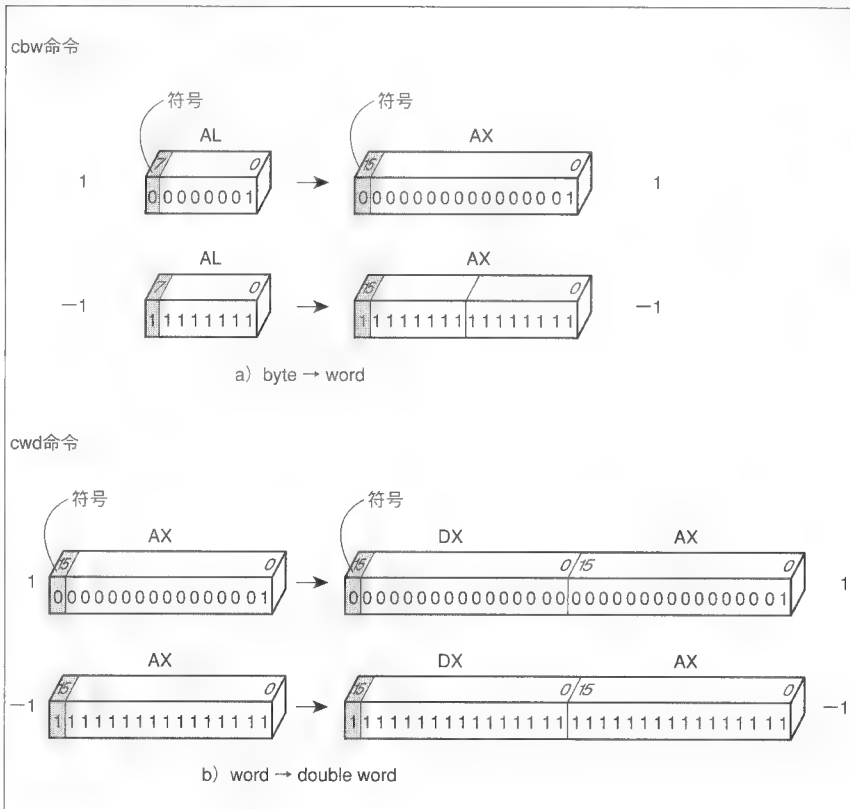
符号付きの除算には，idiv命令を用います．データを符号付きと見なして演算する他はdiv命令と同じ働きとなります．

図4．10●除算命令

divとidiv除算の被除数は，除算前にバイト→ワードへ，ワード→ダブルワードへ変換する必要があります．8086にはこの変換のための命令がありますが，基本的に符号の付いたデータを対象としています．

```
cbw          ;(AL)を(AH:AL)に拡張
cwd          ;(AX)を(DX:AX)に拡張
```

cbwはバイトをワードデータに，cwdはワードデータをダブルワードデータに拡張します．**図4．11**に符号付きデータの拡張を示します．10進数の1は，レジスタAL内では2進数で00000001と表現されます．これはcbw命令によって，AX内で0000000000000001と拡張されます．10進数の－1はAL内で11111111で表現されますが，cbw命令によって1111111111111111と拡張されます．cwdでは同様にAXのデータがDX:AXのダブルワードデータに拡張さ

れます．

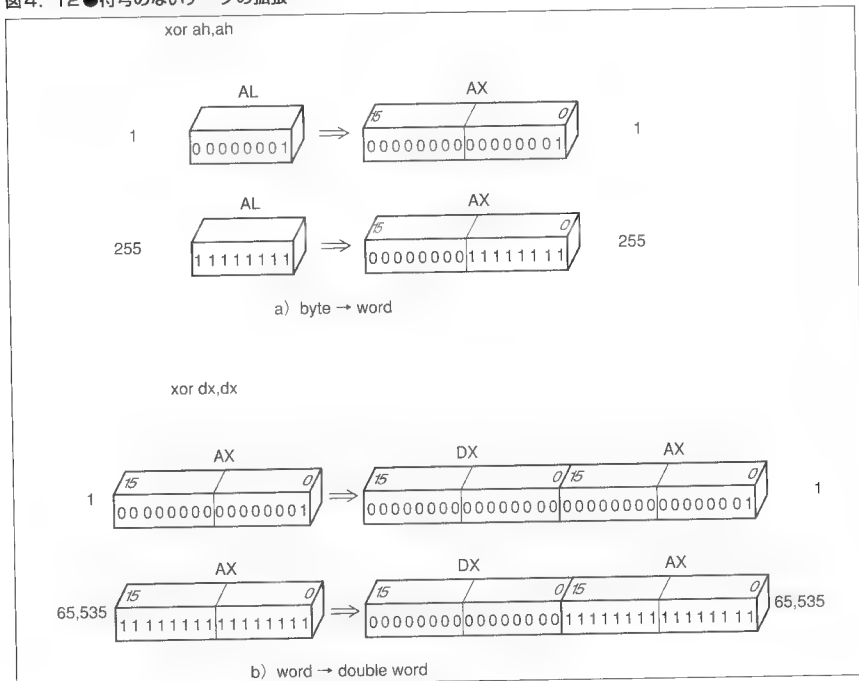
図4．11●符号付きデータの拡張

符号のないデータには，cbwとcwd命令は適用できません．ALレジスタが11111111である場合，符号のないデータでは255を表現していることになります．これをAX(AH:AL)に拡張する場合は，AHレジスタをクリアする必要があります．

```
mov al,255      ;ALに定数255をロード
xor ah,ah       ;AHをクリアして拡張
```

AXレジスタをDX:AXに拡張する場合は，

```
mov ax,255          ;AXに定数255をロード
xor dx,dx           ;DXをクリアして拡張
```

この拡張法を**図4. 12**に示します.

図4. 12●符号のないデータの拡張

idiv命令のプログラム例を**リスト4. 10**に示します. 変数xとyの除算の商を変数dへ, 余りをrに保存し, 確認のために16進表示を行います. ここでもマクロ定義ファイルiomac.incをincludeし, 16進表示のための外部手続きhexWをコールします. 実際の処理部分は行11:〜15:までの命令で, 被除数xをALレジスタにロードし, 符号付きデータとしてcwdでDX:AXへ拡張します. idivで除算した結果の商をdへ, 余りをrへ保存します. 行17:のhexWで外部手続きを引用して除算結果を16進表示します. 25／9の除算結果が実行例に示してあります.

リスト4．10●idiv命令のプログラム（x/yの商と余りを表示）

```
 1: .model small              ;メモリモデルsmall
 2: ;----------------------------
 3: include iomac.inc         ;MS-DOSシステムコールのマクロ定義ファイル
 4: extrn hexW :near          ;axの内容を16進表示する外部手続き
 5: ;----------------------------------------
 6: .code                     ;コードセグメントの始まり
 7: ;
 8: start:      mov ax,@data  ;データセグメントアドレス
 9:             mov ds,ax     ;データセグメントを設定
10: ;
11:             mov ax,x      ;xの値をaxへ移動
12:             cwd           ;axの値をdx:axへ拡張
13:             idiv y        ;(ax)←(dx:ax)/(y)  (dx)←余り
14:             mov d,ax      ;商をdへ保存
15:             mov r,dx      ;余りをrへ保存
16: ;
17:             call hexW     ;商を16進表示(axの内容)
18:             crlf          ;改行(マクロ)
19:             mov ax,r      ;余りをaxへ移動
20:             call hexW     ;余りを16進表示
21: ;
22:             exit          ;MS-DOS復帰(マクロ)
23: ;-------------------------------------
24: .data       ;データセグメントの始まり
25: x           dw 25         ;変数xを初期値25で定義
26: y           dw 9          ;変数yを初期値9で定義
27: d           dw ?          ;商保存用変数．初期化せず
28: r           dw ?          ;余り保存用変数．初期化せず
29: ;-------------------------------------
30: .stack 100H               ;スタックセグメント
31: ;-------------------------------------
32:             end start     ;startから実行開始
```

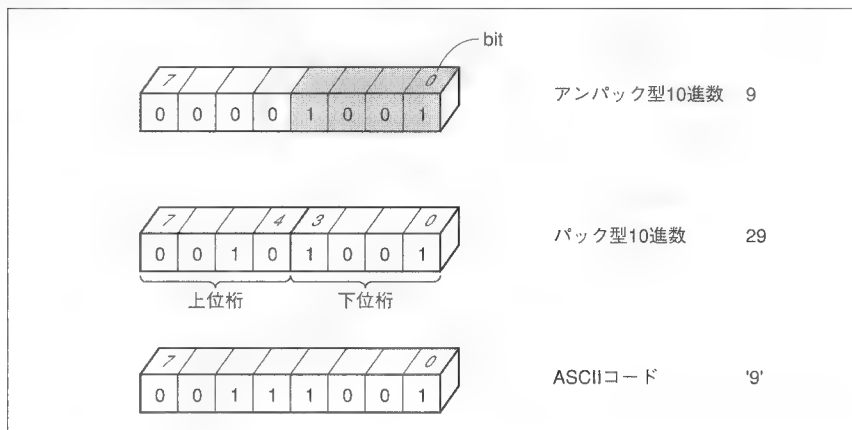
実行例

```
C:¥prg>idiv
0002
0007
```

4.6 10進数の演算

8086のルーツであるマイクロプロセッサ4004や8008は，新しい電卓を作るのを容易にするために開発されました．8086でも電卓が行う10進演算を簡単に行えるように作られた命令群があります．ASCII演算補正命令や10進演算補正命令がそれに当たります．

10進数をコンピュータ内部で表現する場合，2進化10進数(BCD：binary coded decimal)が用いられます．10進数も**図4．13**に示すように，基本的には2進数で表現します．BCDは4ビットで1桁の10進数を表現するため，これを1バイトで表現すると4ビット分の余分を生じます．この余分を無視して1バイトにBCD1桁を表現したものがアンパック型10進数です．これに対して，余った4ビットにもBCD1桁を詰めたものがパック型10進数です．パック型10進数の方がメモリ効率がよいわけですが，アンパック型は1バイトに1桁であるために処理が簡単になる利点があります．aaa，aas，aam，aadはアンパック型の10進演算に，daa，dasはパック型10進演算に適用する補正命令として用意されています．

図4．13● 10進数の内部表現

リスト4．11に10進補正命令の使用例を，**表4．3**に10進補正命令の処理内容を示します．aaa，aas，aam，aadはパック型10進数の補正のための命令で

す．1バイトで表現された2進化10進数(BCD　ASCIIコードでも可)の演算前または後に，BCDの範囲を超えた値をBCDに補正するための命令です．加算，減算，乗算では演算前に，除算では後に補正を行います．これらの演算結果はALレジスタにあることを前提としています．

これに対してパック型10進数の演算後に，BCDの範囲を超えた値をパック型BCD表現に補正するための命令がdaaとdasです．

第4章 算術演算命令

リスト4．11●10進補正命令

```
 1: .code
 2:      adc al,bl              ;加算
 3:      aaa                    ;ASCII(アンパック)加算後の補正
 4: ;
 5:      sbb al,bl              ;減算
 6:      aas                    ;ASCII(アンパック)減算後の補正
 7: ;
 8:      mul bl                 ;乗算
 9:      aam                    ;ASCII(アンパック)乗算後の補正
10: ;
11:      aad                    ;ASCII(アンパック)除算前の補正
12:      div bl                 ;除算
13: ;
14:      adc al,bl              ;加算
15:      daa                    ;10進(パック)加算後の補正
16: ;
17:      sbb al,bl              ;減算
18:      das                    ;10進(パック)減算後の補正
```

10進補正命令の具体的な処理内容を**表4．3**に示します．

次に，具体的な10進演算の例を示します．ここでも，MS-DOSシステムコールは前述のように9章2節で定義するマクロ命令を使用します．また，10進数表示にはプログラム中でddispとして定義するマクロ命令を用います．

表4．3●10進補正命令の処理内容

命令	オペランド	動作内容	機能
AAA		if ((AL) & 0FH)>9 or (AF)=1 then (AL)←(AL)+6, (AH)←(AH)+1,(AF)←1,(CF)←(AF),(AL)←(AL) & 0FH	ASCII加算後の補正
AAD		(AL)←(AH)*0AH+(AL), (AH)←0	ASCII除算前の補正
AAM		(AH)←(AL)/0AH, (AL)←(AL)%0AH	ASCII乗算後の補正
AAS		if ((AL) & 0FH)>9 or (AF)=1 then (AL)←(AL)-6, (AH)←(AH)-1,(AF)←1,(CF)←(AF),(AL)←(AL) & 0FH	ASCII減算後の補正

DAA		if ((AL) & 0FH)>9 or (AF)=1 then
		(AL)←(AL)+6,(CF)←(AF)∨(CF) ,(AF)←1
		if ((AL) >9FH or (CF)=1 then (AL)←(AL)+60H, (CF)←1
			10進加算後の補正

DAS		if((AL) & 0FH)>9 or (AF)=1 then (AL)←(AL)-6,
		,(CF)←(AF)∨(CF) ,(AF)←1
		if((AL) >9FH or (CF)=1 then (AL)←(AL)-60H, (CF)←1
			10進減算後の補正

( )　　レジスタまたはメモリの内容
(( ))　　レジスタまたはメモリが指すメモリの内容

■アンパック型10進数の加算

プログラム例を**リスト4．12**に示します．データ領域にある変数xの'9'とyの'2'のASCIIデータを加算後補正して表示する簡単な例です．

行3:でMS-DOSシステムコールを行うマクロ定義ファイルiomac.incをインクルッド(include)します．行7:～11:ではBCD1桁を表示するddispを，内部でマクロ定義します．マクロについては9章2節を参照してください．

行34:～37:はデータ設定で，被加数xをASCIIコード'9'に，加数yを'2'に初期化します．行18:～23:までが実際の処理部分です．まずAHレジスタとキャリフラグCFをクリアして桁上げに備えます．ALレジスタに被加数x('9')をロードし，次のadc al,yで加数y('2')と加算しています．ここでaddでなくadcを用いて，キャリフラグCFも加算しているのは，桁数が複数になった場合を想定しています．ここでは，add命令を用いても同じ結果が得られます．この後aaa命令によって，加算後の補正を行います．

aaaの処理を**図4．14**に示します．この例題では被加数，加数共に1桁ですが，AHレジスタに被加数の上位桁があるとみなして処理されます．adc al,yによる計算結果はALレジスタに保存されます．このALの下位4ビットが10以上になっていれば，BCDの範囲を超えていることになります．そこでaaaによって，

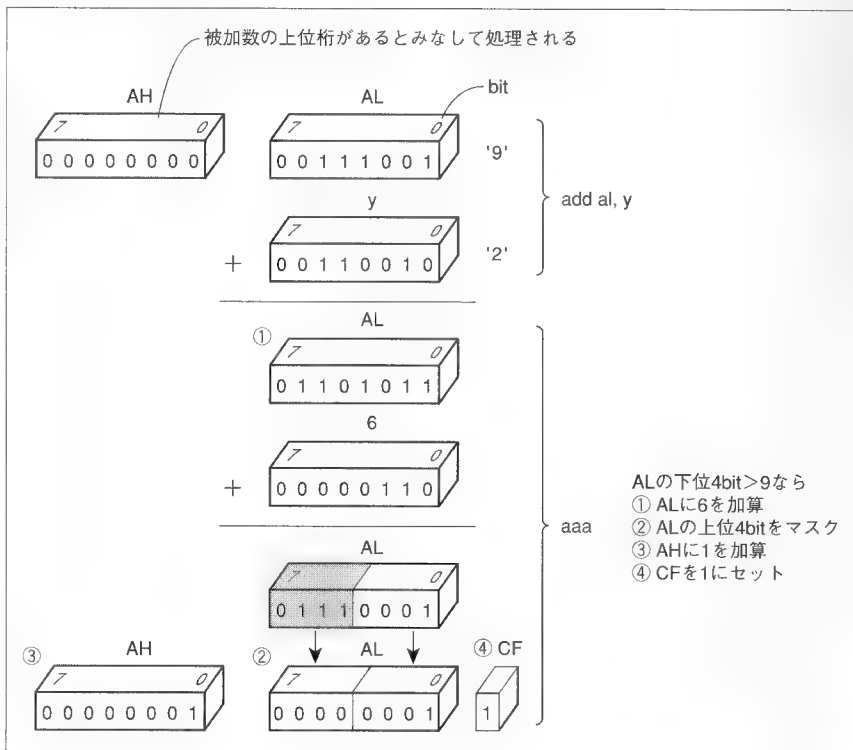
①ALに6を加算

②ALの上位4ビットをマスク

③AHに1を加算

④CFを1にセット

を行います．ALに6を加算することで，10～15までの数は上位4ビットに桁上げされ，下位BCDは9以下になります．これにより生じた桁上げがAHとCFに設定されます．プログラム例では1桁の加算ですからAHに，計算結果の上位桁が求まったことになります．

図4．14●aaaの処理

行26:～30:までは，計算結果を表示するプログラムです．BXレジスタへ計算結果をロードし，BCD1桁を表示するマクロddispを2度呼んでいるだけです．マクロ展開されたリストはわかりにくくなるため示していませんが，アセンブル

すると，次のように展開されます．

```
ddisp bh                        ;上位桁を10進表示(マクロ)
        mov dl,bh               ;展開されたコード
        add dl,'0'              ;展開されたコード
        disp                    ;1文字表示(マクロ)
                mov ah,2        ;展開されたコード
                int 21h         ;展開されたコード
```

実行例では，9＋2という計算をした結果として11が表示されています．11の後ろに付いているDは，10進表示であることを示すものです．

リスト4．12●アンパック型10進数の加算（'9'+'2'の結果を表示）

```
 1: .model small
 2: ;--------------------------
 3:      include iomac.inc    ;MS-DOSシステムコールのマクロ定義ファイル
 4: ;-------------------------------------
 5: ;マクロ定義
 6: ;
 7: ddisp macro  reg          ;マクロ定義(10進表示)
 8:              mov dl,reg   ;表示データをDLへ
 9:              add dl,'0'   ;数字0を加算→数字化
10:              disp         ;1桁を表示(マクロ参照)
11:              endm
12: ;-------------------------------------
13: .code        ;コードセグメント
14: start:
15:              mov ax,@data ;@dataはセグメントアドレス
16:              mov ds,ax    ;DSへデータセグメントの設定
17: ;--------------------------
18:              xor ah,ah    ;AHをクリア
19:              clc          ;キャリフラグをクリア
20:              mov al,x     ;被加数x
21:              adc al,y     ;x+y+CF
22:              aaa          ;アンパック10進補正.桁上げをCFへセット
```

```
23:             mov ans,ax      ;計算結果を保存
24: ;------------------------
25:                             ;BCD桁を10進表示
26:             mov bx,ans      ;計算結果をBXへ
27:             ddisp bh        ;上位桁を10進表示(マクロ参照)
28:             ddisp bl        ;下位桁を10進表示(マクロ参照)
29:             mov dl,'D'      ;10進数の後にDを表示
30:             disp            ;Dを表示
31: ;
32:             exit            ;MS-DOS復帰(マクロ参照)
33: ;------------------------------------------
34: .data           ;データセグメント
35: x   db '9'                  ;被加数
36: y   db '2'                  ;加数
37: ans dw  ?                   ;加算結果
38: ;------------------------------------------
39: .stack 100H                 ;スタックセグメント
40: ;------------------------------------------
41:       end start             ;startからプログラムを開始することを示す
```

実行例

```
C:¥prg>ex_dadd
11D
```

■アンパック型10進数の減算

アンパック型10進数の減算プログラム例を**リスト4. 13**に示します．加算の前例題を減算に変更した例で，データ領域にある変数xの'9'とyの'2'のASCIIデータを減算後，補正して表示する簡単な例です．

行7:～11:でのマクロ命令の定義を行っています．行18:～23:までが実際の処理部分で，まずAHレジスタとキャリフラグCFをクリアをして，桁借りに備えます．行20:でALレジスタに被減数x('9')をロードし，次のsbb al,yで減数y('2')と減算しています．ここでsubでなくsbbを用いて，キャリフラグCFも減算しているのは，桁数が複数になった場合を想定しています．ここでは，sub命令を用いても同じ結果が得られます．この後aas命令によって，減算後の補正を行います．

aasの処理を説明します．aasの処理でも前例題同様に，AHレジスタに被減

数の上位桁があるとみなして処理されます．sbb al,yによる計算結果はALレジスタに保存されます．このALの下位4ビットが10以上になっていれば，上位桁からの借りがあったことでBCDの範囲を超えていることになります．そこでaasによって，

①ALから6を引く

②ALの上位4ビットをマスク

③AHから1を引く

④CFを1にセット

を行います．これによりALに減算結果が得られたことになります．CFは桁借りがあったことを示しています．

行26:～30:は前例題と同様に，計算結果を表示するプログラムです．実行例に示すように，9－2という計算をした結果として07が表示されています．

リスト4．13●アンパック型10進数の減算（'9'－'2'の結果を表示）

```
 1: .model small
 2: ;-------------------------
 3:      include iomac.inc    ;MS-DOSシステムコールのマクロ定義ファイル
 4: ;-----------------------------------
 5: ;マクロ定義
 6: ;
 7: ddisp macro  reg          ;マクロ定義(10進表示)
 8:              mov dl,reg   ;表示データをDLへ
 9:              add dl,'0'   ;数字0を加算→数字化
10:              disp         ;1桁を表示(マクロ参照)
11:              endm
12: ;-----------------------------------
13: .code                     ;コードセグメント
14: start:
15:              mov ax,@data ;@dataはセグメントアドレス
16:              mov ds,ax    ;DSへデータセグメントの設定
17: ;-------------------------
18:              xor ah,ah    ;AHをクリア
19:              clc          ;キャリフラグをクリア
20:              mov al,x     ;被減数x
21:              sbb al,y     ;x-y-CF
22:              aas          ;アンパック10進補正.借りをCFへセット
23:              mov ans,ax   ;計算結果を保存
24: ;-------------------------
```

```
25:                          ;BCD桁を10進表示
26:            mov bx,ans    ;計算結果をBXへ
27:            ddisp bh      ;上位桁を10進表示(マクロ参照)
28:            ddisp bl      ;下位桁を10進表示(マクロ参照)
29:            mov dl,'D'    ;10進数の最後のD
30:            disp          ;Dを表示
31: ;
32:            exit          ;MS-DOS復帰(マクロ参照)
33: ;---------------------------------------------
34: .data       ;データセグメント
35: x   db '9'                ;被減数
36: y   db '2'                ;減数
37: ans dw  ?                 ;減算結果
38: ;---------------------------------------------
39: .stack 100H               ;データセグメント
40: ;---------------------------------------------
41:       end start           ;startからプログラムを開始することを示
```

実行例
```
C:¥prg>dsub
07D
```

■アンパック型10進数の乗算

プログラムの例を**リスト4. 14**に示します．加算，減算と同じく，データ領域にある変数xの'9'とyの'2'のASCIIデータを乗算後補正して表示する例です．

行18:～23:までが実際の処理部分で，まずALレジスタに被乗数x('9')をロードし，0FHとAND(5章参照)をとって上位4ビットをマスクし，ASCIIコードをBCDに変換します．さらに乗数yも上位4ビットをマスクし，BCDに変換します．加減算と異なり，乗算では上位ビットが計算結果の問題となるため[illegible]

次のmul yで乗数yと乗算されます．

この後，aam命令によって，乗算後の補正を行います．aamの処理[illegible]AHレジスタに乗算結果の上位桁(BCD)が，ALレジスタに下位桁(BCD)が[illegible]ります．aamによって，

①ALを10で割り，　商をAHへセット　上位BCD

②　　　　　　　　余りをALへセット　下位BCD

が行われます．これによりAH:ALに乗算結果の上下桁(BCD)が得られ[illegible]

なります.

行26:〜30:は前例題と同様に，計算結果を表示する部分で，実行例に示すように，9×2という計算をした結果として18が表示されています.

リスト4．14●アンパック型10進数の乗算（9×2の結果を表示）

```
 1: .model small
 2: ;--------------------------
 3:       include iomac.inc    ;MS-DOSシステムコールのマクロ定義ファイル
 4: ;-----------------------------------
 5: ;マクロ定義
 6: ;
 7: ddisp macro  reg           ;マクロ定義(10進表示)
 8:              mov dl,reg    ;表示データをDLへ
 9:              add dl,'0'    ;数字0を加算→数字化
10:              disp          ;1桁を表示(マクロ参照)
11:              endm
12: ;-----------------------------------
13: .code        ;コードセグメント
14: start:
15:              mov ax,@data  ;@dataはセグメントアドレス
16:              mov ds,ax     ;DSへデータセグメントの設定
17: ;--------------------------
18:              mov al,x      ;被乗数x
19:              and al,0fh    ;ASCII数字をBCDへ
20:              and y,0fh     ;乗数yを数字からBCDへ
21:              mul y         ;x*yの結果がAX
22:              aam           ;アンパック10進補正.桁上げをCFへセット
23:              mov ans,ax    ;計算結果を保存
24: ;--------------------------
25: ;BCD桁を10進表示
26:              mov bx,ans    ;計算結果をBXへ
27:              ddisp bh      ;上位桁を10進表示(マクロ)
28:              ddisp bl      ;下位桁を10進表示(マクロ)
29:              mov dl,'D'    ;10進数の最後のD
30:              disp          ;Dを表示
31: ;
32:              exit          ;MS-DOS復帰(マクロ)
33: ;-------------------------------------------
34: .data        ;データセグメント
35: x   db '9'                 ;被乗数
36: y   db '2'                 ;乗数
37: ans dw  ?                  ;乗算結果
```

```
38: ;-----------------------------------------------
39: .stack 100H             ;スタックセグメント
40: ;-----------------------------------------------
41: end start               ;startからプログラムを開始することを示す
```

実行例

```
C:¥prg>dmul
18D
```


■アンパック型10進数の除算

プログラムの例をリスト4.15に示します．データ領域にある変数xの'9'を除算前に補正し，yの'2'で除算して表示する例です．

行18:～24:が実際の処理部分で，まずALレジスタに被除数x('9')をロードし，0FHとAND(5章参照)をとって上位4ビットをマスクし，ASCIIコードをBCDに変換します．さらにワード化するためにAHレジスタをクリアします．除算では，除算前にaad命令で補正を行います．aadの処理では，AHに上位桁(BCD)があると想定して，この値に10をかけた値と，ALの下位桁(BCD)を加算してALの値とします．これはBCDコードを純2進数に変換したことを意味します．aadによって，

①AHに10を乗じてからALと加算，結果をALへセット(純2進化)

②AHをクリア

が行われます．これによりAH:ALに純2進化した値が得られます．

次に除数yも上位4ビットをマスクし，BCDに変換します．div yで，AH:ALの値が除数yによって除算されます．結果は商がALに，余りがAHに残るので，これを変数ansに保存します．この例題では1桁同士の除算ですが，複数桁の10進除算では除算後に，さらに補正が必要になる場合があります．

行27:～31:は計算結果を表示する部分で，実行例に示すように，9／2の計算をした余りと商が14として表示されています．1が余りで，4が商を示しています．

リスト4．15●アンパック型10進数の除算（'9'／'2'の結果を表示）

```
 1: .model small
 2: ;-------------------------------------
 3:       include iomac.inc    ;MS-DOSシステムコールのマクロ定義ファイル
 4: ;-------------------------------------
 5: ;マクロ定義
 6: ;
 7: ddisp  macro reg           ;マクロ定義(10進表示)
 8:             mov dl,reg     ;表示データをDLへ
 9:             add dl,'0'     ;数字0を加算→数字化
10:             disp           ;1桁を表示(マクロ参照)
11:             endm
12: ;-------------------------------------
13: .code                      ;コードセグメント
14: start:
15:             mov ax,@data   ;@dataはセグメントアドレス
16:             mov ds,ax      ;DSへセグメントアドレスの設定
17: ;-------------------------------
18:             mov al,x       ;被除数x
19:             and al,0fh     ;ASCII数字をBCDへ
20:             xor ah,ah      ;word化
21:             aad            ;アンパック10進補正
22:             and y,0fh      ;除数yを数字からBCDへ
23:             div y          ;x/yの商がAL，余りがAH
24:             mov ans,ax     ;計算結果を保存
25: ;-------------------------------
26:                            ;BCD桁を10進表示
27:             mov bx,ans     ;計算結果をBXへ
28:             ddisp bh       ;上位桁を10進表示(マクロ参照)
29:             ddisp bl       ;下位桁を10進表示(マクロ参照)
30:             mov dl,'D'     ;10進数の最後のD
31:             disp           ;Dを表示
32: ;
33:                 exit           ;MS-DOS復帰(マクロ参照)
34: ;------------------------------------------------
35: .data                      ;データセグメント
36: x   db '9'                 ;被除数
37: y   db '2'                 ;除数
38: ans dw ?                   ;計算結果  余：商
39: ;------------------------------------------------
40: .stack 100H                ;データセグメント
41: ;------------------------------------------------
42:      end start                 ;startからプログラムを開始することを示す
```

実行例

```
C:¥prg>dmul
14D
```

演習問題4

1. 次の算術演算命令を記述せよ．データはデータセグメント，コードはコードセグメントに分けること．

(1) byte変数xとyを加算し，byte変数aへ保存する．
(2) word変数mからnを減算し，word変数aに保存する．
(3) word変数xとyの値を，一方をSIレジスタを用いた間接アドレスで加算し，word変数aへ保存する．
(4) word変数xとyの加算をbyte単位で行い，結果をword変数aへ保存する．
(5) word変数xとyの減算をbyte単位で行い，結果をword変数aへ保存する．
(6) byte変数bから1を減じて，byte変数aへ保存する．
(7) word変数wを1増して，word変数aへ保存する．
(8) byte変数bとword変数wの乗算結果を，word変数hとlに保存する．無符号データとする．
(9) word変数aとword変数bの除算結果をxに，余りをyに格納する．有符号データとする．
(10) byte変数bの値を5倍して，word変数wに保存する．無符号データとする．
(11) byte変数x（1の位）とy（10の位）のBCDコードを，1バイトの純2進コードに変換し，word変数wへ保存する．
(12) byte変数xの数字'7'とyの'5'を加算し，BCDコードとしてword変数aへ保存する．アンパックデータとする．
(13) byte変数xの数字'9'からyの'3'を減算し，BCDコードとしてbyte変数aへ保存する．アンパックデータとする．
(14) byte変数xのBCDコード5とyの3を乗算し，結果をBCDコードとしてword変数aへ保存する．アンパックデータとする．

2. 次のプログラムを記述し，アセンブルして実行せよ．

(1) 変数xとyに10以下の適当な初期値を与えて，加算した結果を数字1桁で表示する．
(2) 2つの数字(例31)をキーボードから入力し，減算した結果を表示する．
(3) キーボードから1文字入力し，これをインクリメントして表示する．英字と数字について試すこと．
(4) キーボードから1文字入力し，これをディクリメントして表示する．英字と数字について試すこと．

第 5 章

ビット操作命令

ビット操作命令は，ビット単位に演算や処理を行う命令で，論理演算，論理比較，シフト，ローテート命令があります．論理演算命令には，and, or, xor,not 演算があり，ビット単位で演算を行います．また論理比較のために，test命令があります．その他シフト命令として，shl/sal，shr，sar命令があり，ローテート命令としてrol，ror，rcl，rcr命令があります．

なお，この章のプログラム例では，レジスタAXの内容を16進数表示するためのサブルーチンhexWを用います．このサブルーチンは6章5節で定義するもので，外部手続きとして宣言してから使用します．サブルーチンについてはまだ学んでいませんが，表示ツールとして使用します．

5.1 論理演算命令

表5．1に論理演算命令の一覧を示します．論理演算命令はand, or, xor,notを行う命令ですが，オペランドは算術演算命令と同じものが許されます．ただし，not命令はオペランドは1つだけを指定します．

表5．1●論理演算命令一覧

	命令	機能	
論理演算	NOT	"not" byte or word	否定
	AND	"and "byte or word	論理積
	OR	'or" byte or word	論理和
	XOR	'exclusive or " byte or word	排他的論理和
論理比較	TEST	logical test byte or word	論理積をとってフラグをセット

基本演算の真理値表を，**表5．2**に示します．演算される各ビットをAとBで示し，これにand, or, xor,not演算を行った場合の結果を示してあります．

表5．2●基本演算の真理値表

論理積

A	B	and
0	0	0
0	1	0
1	0	0
1	1	1

論理和

A	B	or
0	0	0
0	1	1
1	0	1
1	1	1

排他的論理和

A	B	xor
0	0	0
0	1	1
1	0	1
1	1	0

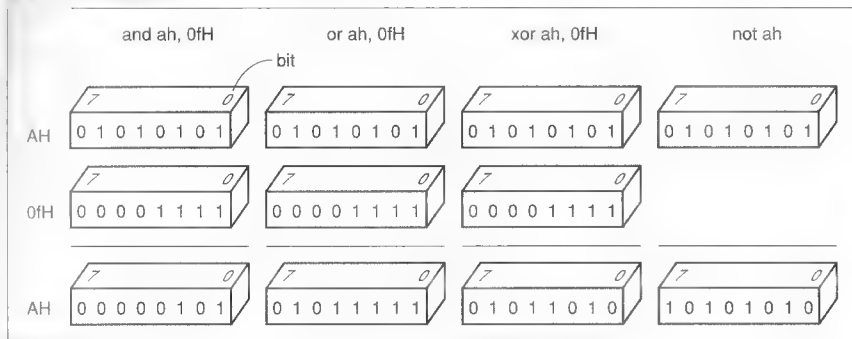
否定

A	not
0	1
1	0

図5．1に，論理演算命令がどのようにビット単位で行われるかを示します．AHレジスタにデータ01010101(55H)があり，直接データとして00001111(0fH)を用いています．例としての論理演算命令を下に示します．

```
and ah,0fH      ;(AH)←(AH)・0fH
or ah,0fH       ;(AH)←(AH)+0fH
xor ah,0fH      ;(AH)←(AH)⊕0fH
not ah          ;(AH)←(AH)  (AH上線)
```

and命令はソースとディスティネーションオペランドの各ビットごとにand演算を行います．and ah,0fHでは，各ビットごとにand演算が行われ，演算結果が得られます．or，xor，not演算についても同様に示してあります．

図5．1●論理演算命令

リスト5．1に論理演算命令の使用例を示します．論理演算命令でも，

```
and    destination,source
```

のように，最初のオペランドがディスティネーションで，2番目のオペランドがソースとなり，演算結果がディスティネーションに保存されます．

行2:のand ax,bxでは16ビットのレジスタAXとBXの論理積がビット単位でとられ，結果がAXレジスタへ保存されます．行3:のor al,0fHではALと直接データ0fHでor演算が行われます．行4:ではBXと変数wとの排他的論理和がとられ，結果がBXへ保存されます．行5:は変数xの否定，結果がxに保存されます．

行6:のtest命令では，変数wとDXとの論理積がとられますが，結果はどこにも保存されません．この演算で生じた状態で，フラグレジスタをセットするためにある命令で，算術演算のcmp命令と働きが似ています．

行7:のmov命令は間接アドレスに用いるために，変数xのアドレスをBXレジスタへに設定しています．行8:ではALとBXで間接指定したメモリアドレスの内容とでandがとられます．BXにはxのアドレスが与えられているので，間接的にxの値をALレジスタへ渡すことになります．

リスト5．1●論理演算命令

```
 1: .code                    ;コードセグメントの先頭
 2:      and ax,bx           ;axとbxの論理積
 3:      or  al,0fH          ;alと0fとの論理和
 4:      xor bx,w            ;bxと変数wとの排他的論理和
 5:      not x               ;変数xの否定
 6:      test w,dx           ;変数wとdxとの論理積でフラグを設定
 7:      mov bx,offset x     ;変数xのアドレスをbxに設定
 8:      and al,[bx]         ;alとbx間接でメモリ上のデータと論理積
 9: .data                    ;データセグメントの先頭
10: x    db ?                ;word変数x
11: w    dw ?                ;word変数w
```

xor命令はレジスタやメモリをクリアする場合よく用いられます．たとえばレジスタALを0にするには，

mov al,0

としてもよいわけですが，次のようにすると簡単で実行速度は速くなり，コード(機械語)メモリの効率化も得られます．

xor al,al

このxor命令では，2つのオペランドが共にALを指定してあるため，演算の結果すべてのビットが0となります．xor演算は2つのオペランドの対応するビットが異なった値であるとき結果が1になるので，まったく同じレジスタを指定した場合は全ビットが必ず0となります．

xor命令のプログラム例を**リスト5．2**に示します．データ領域に配列x(要素数20H)をとり，この領域を0にクリアするプログラム例です．行29:のデータセグメントにある領域定義

x db 20h dup(?)

では，配列xを要素数20H(32_{10})で，dup(?)によって初期化しないことを宣言しています．dupがあるために，変数xの初期値を20Hするのではないことを意味しています．

行9:では，カウンタとして用いるCLレジスタにデータ数20Hを与えます．次にDIレジスタへ配列xのアドレスを設定し，行11:でALレジスタをクリアします．

行12:～15:で実際に配列xの全要素を0にクリアします．行12:でクリアされたALレジスタの内容を，DIレジスタを間接アドレスに用いて，配列xの要素に与えます．その後，要素数が設定されたCLレジスタをディクリメントして，CLが0になっていないなら，ラベルLPへジャンプします．jnz命令はまだ学んでいない命令ですが，dec命令でCLが0にならなかった場合に，ラベルlpへジャンプしてlpからの命令を再実行するものです．したがってこのループは32(20H)回繰り返されることになります．

行18:～24:までは，配列xの内容が実際に0にクリアされたかを確認するためのループです．この部分が実行されると，ディスプレイに32個(20H個)の0が表示されます．もし表示されない場合は，どこかにプログラムの誤りがあるはずです．行26:では，MS-DOSへの復帰が起こります．

リスト5．2●xor命令のプログラム例（配列xをクリアし、表示する）

```
 1: .model       small           ;メモリモデルsmall
 2: ;-----------------------------------------------------
 3: include iomac.inc            ;MS-DOSシステムコールのマクロ定義ファイル
 4: ;
 5: .code                        ;コードセグメントの始まり
 6: start: mov ax,@data          ;データセグメントアドレス
 7:        mov ds,ax             ;データセグメントを設定
 8: ;
 9:        mov cl,20h            ;clに20hを設定
10:        mov di,offset x       ;配列xのアドレスをdiへ
11:        xor al,al             ;alをクリア
12: lp:    mov [di],al           ;alの内容0をdi間接でxへ転送
13:        inc di                ;配列xのアドレスを1増
14:        dec cl                ;clを1減
15:        jnz lp                ;dec結果が0でないならlpへジャンプ
16: ;
17:        mov cl,20h            ;clに20hを設定
18:        mov di,offset  x      ;配列xのアドレスをdiへ
19: lpdsp: mov dl,[di]           ;di間接でxの内容をdiへ転送
20:        add dl,'0'            ;数字にするため、文字0を加算
21:        disp                  ;文字表示(マクロ)
22:        inc di                ;配列xのアドレスを1増
23:        dec cl                ;カウンタを1減
24:        jnz lpdsp             ;dec結果が0でないならlpdspへ
25: ;
26:        exit                  ;MS-DOS復帰(マクロ)
```

```
27: ;-----------------------------------
28: .data                              ;データセグメントの始まり
29: x       db  20h  dup(?)            ;配列x 初期化せず
30: ;-----------------------------------
31: .stack 100h                        ;スタックセグメント
32: ;-----------------------------------
33:         end start                  ;startから実行開始
```

実行例

```
C:¥prg>clear
00000000000000000000000000000000
```

5.2 シフト／ローテート命令

表5．3にシフト／ローテート命令を示します．シフト命令とローテート命令は，レジスタやメモリ内のデータを右または左方向へビット単位で移動させる命令です．シフトでは移動してレジスタ外へ飛び出したデータが消えてしまうのに対して，ローテート命令は移動してレジスタ外へ飛び出したデータは回転する形で，レジスタ内に戻ることになります．シフト／ローテート命令はビット単位でデータを処理する場合に用います．

表5．3●シフト／ローテート命令一覧

	命令	機能	
シフト	SHL/SAL	shift logical/arithmetic left byte or word	左シフト／算術左シフト
	SHR	shift logical right byte or word	右シフト
	SAR	shift arithmetic right byte or word	算術右シフト
ローテート	ROL	rotate left byte or word	左ローテート
	ROR	rotate right byte or word	右ローテート
	RCL	rotate thru carry left byte or word	キャリフラグを通した左ローテート
	RCR	rotate thru carry right byte or word	キャリフラグを通した右ローテート

リスト5．3にシフト／ローテート命令の使用例を示します．1ビットシフト／ローテートする場合は，ソースオペランドに定数1を記述します．ここに2以上の数を書くことはできません．複数ビットの場合は，CLレジスタにシフト／ローテート数を与え，ソースオペランドにCLと記述します．これによりCLがカウンタとして用いられ，与えた数だけ自動的にシフト／ローテートされます．

行2:ではシフト／ローテートのビット数を決めるため，カウンタとして用いるCLレジスタを7に設定しています．行4:のshl al,1は，ALレジスタを1ビット左へ論理シフトする例です．このシフト数は，1だけが許されていて，shl al,3という記述は許されません．

行5:のshr bx,clでは，BXレジスタに対して，CLに与えた数だけ右シフトが行われます．salとsarは算術的に左右のシフトをする例です．ただしsal命令はshl命令とまったく同じ結果となります．

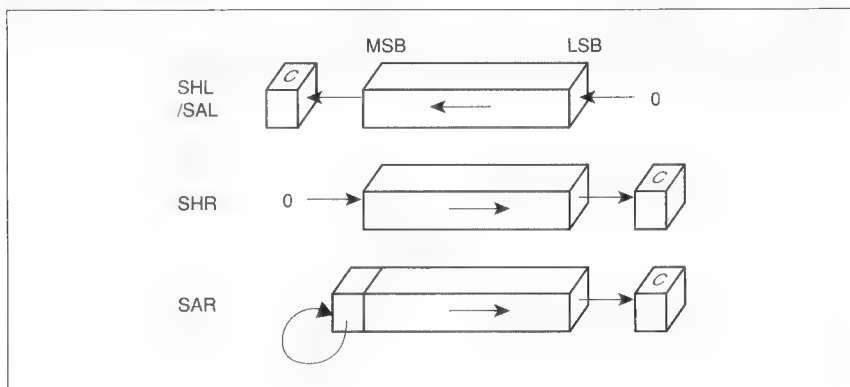
行9:〜12:はローテート命令の例です．記述方法はシフト命令の場合と同じく，1ビットローテートの場合は，rol x,1のようにローテートするビット数'1'を直接データとして記述します．2ビット以上ローテートする場合は，カウンタとしてCLレジスタを用いることになります．

リスト5．3●シフト／ローテート命令の使用例

```
 1: .code                   ;コードセグメントの先頭
 2:     mov cl,7            ;カウンタCLに7を設定
 3: ;
 4:     shl al,1            ;alを1ビット左へ論理シフト
 5:     shr bx,cl           ;bxをカウンタCLにより，右へ論理シフト
 6:     sal w,1             ;変数wを1ビット左へ，算術シフト
 7:     sar x,cl            ;変数xをカウンタCLにより，右へ算術シフト
 8: ;
 9:     rol x,1             ;変数xを1ビット，左ローテート
10:     ror ax,cl           ;axをカウンタclにより，右ローテート
11:     rcl ax,1            ;axを1ビット，フラグCFを通して，左ローテート
12:     rcr bl,cl           ;blをカウンタclにより，フラグCFを通して右へ，ローテート
13: .data                   ;データセグメントの先頭
14: x   db  ?               ;word変数x
15: w   dw  ?               ;word変数w
```

実際のシフト命令の動きを**図5．2**に示します．shl命令は左(left)方向へビット単位でデータを移動します．1ビットshlする場合，MSBのデータはレジスタ外へ押し出されて，フラグCF(carry flag)にセットされ，MSBへは0がシフトされた形となります．

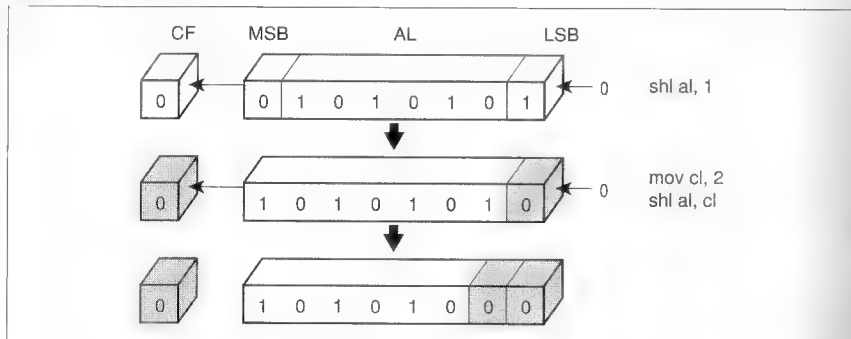
shr(shift right)命令は，右方向にシフトする命令でMSBに0が挿入され，LSBから飛び出すビットはフラグCFへ移動します．sar(shift arithmetic right)命令は，MSBの符号ビットはそのまま保存され，MSB－1以下のビットへ右シフトが生じます．なお，sal(shift arithmetic left)命令の動作はshl命令とまったく同じものとなります．

図5．2●シフト命令の動き

shl命令で実際にどのようにビットデータが移動するかを，**図5．3**に示します．たとえば，レジスタALに2進データ01010101があるとき，shl al,1が実行されると，結果は10101010になりCFには0がセットされ，LSBには0が挿入されます．ここでソースオペランドの直接データ1は，シフトするビット数を表します．しかしここには1しか記述することができず，2ビット以上シフトする場合はCLレジスタを用いてその数値を設定します．10101010となったALに，

mov cl,2

shl al,cl

を実行すると，ALの内容は10101000となります．LSB側の2ビットの0は，このシフトによって挿入された0です．

図5．3●shl命令の結果

shl命令のプログラム例を**リスト5．4**に示します．変数xの内容を左シフトして変数yへ保存する簡単なプログラムです．確認のためにシフト前と後のデータを表示するため，16進表示サブルーチンhexWを用います．hexWは6章5節で定義するもので，外部手続き宣言をして使用します．

行24:～25:で，データ領域に変数xを0A0Fで初期化し，結果を保存する変数yを定義しますが初期化は行いません．

行10:～13:が実際のシフト部分です．カウンタCLにシフト数4を与え，初期値を保存するため変数xの値をレジスタAXへロードします．shl ax,clで左シフトし，これをyに保存します．

行15:～19:はシフト前後のデータを表示するためサブルーチンhexWを引用しています．実行結果に示すように，0A0Fのデータが左4ビットのシフトでA0F0になったことがわかります．

リスト5．4●shl命令のプログラム例（xを左シフトし，表示する）

```
 1: .model   small               ;メモリモデルsmall
 2: ;----------------------------------------------------
 3:   include iomac.inc          ;MS-DOSシステムコールのマクロ定義ファイル
 4:   extrn hexW :near           ;axの内容を16進表示する外部手続き
 5: ;
 6: .code                        ;コードセグメントの始まり
 7: start:mov ax,@data           ;データセグメントアドレス
 8:       mov ds,ax              ;データセグメントを設定
 9: ;
10:       mov cl,4               ;カウンタclに4を設定
11:       mov ax,x               ;変数xの値をaxへ
12:       shl ax,cl              ;axをclの値だけ、論理左シフト
13:       mov y,ax               ;シフト結果を変数yへ保存
14: ;
15:       mov ax,x               ;シフト前の値をxへ移動
16:       call hexW              ;16進表示(axの内容)
17:       crlf                   ;改行(マクロ)
18:       mov ax,y               ;シフト結果yをaxへ移動
19:       call hexW              ;16進表示(axの内容)
20: ;
21:       exit                   ;MS-DOS復帰(マクロ)
22: ;------------------------------
23: .data                        ;データセグメントの始まり
24: x     dw 0a0fH               ;変数x  初期値0a0f
25: y     dw ?                   ;変数y  初期化せず
26: ;------------------------------
27:       end start              ;startから実行開始
```

実行例

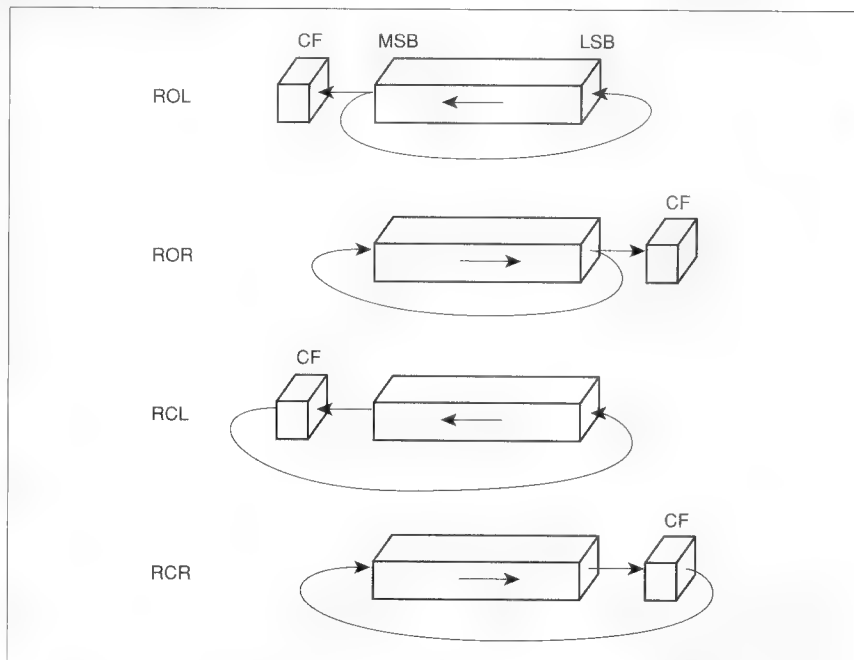
```
C:¥prg>ex_shl
0A0F
A0F0
```

図5．4にローテート命令の動作を示します．ローテート命令には，データだけをローテートさせる命令rolとrorと，データをフラグCFを通してローテートさせる命令rclとrcrがあります．

図に示すようにrol(rotate left)命令では，MSBのデータはLSBへ移動し，その他のビットは左へ1ビットずつ移動します．同時にMSBデータはCFへもセットされます．ror(rotate right)は，LSBデータはMSBへ移動し，その他のビ

ットは右へ回る形で，1ビットずつ移動します．

rcl(rotate thru carry left)命令は，図に示すようにMSBデータがCFへ移動し，CFの値がLSBへ移動します．他のビットは左方向へ1ビットずつ移動します．rcr(rotate thru carry right)は，LSBがCFへ移動，CFの値がMSBへ移動，他のビットは右方向へ回転する形で1ビットずつ移動します．

図5．4●ローテート命令

ror命令のプログラム例を**リスト5．5**に示します．変数xの内容を右ローテートして，変数yへ保存する簡単なプログラムです．確認のためにローテート前と後のデータを表示するため，6章5節で定義する16進表示サブルーチンhexWを用います．

行24:〜25:で，データ領域に変数xを1234Hで初期化し，結果を保存する変数yを定義しますが，初期化は行いません．行10:〜13:が実際のローテート部分で，カウンタCLにローテート数8を与え，初期値を保存するため変数xの値をレジスタAXへロードします．ror ax,clで右回転し，これをyに保存します．

行15:～19:はシフト前後のデータを表示するためサブルーチンを引用しています．実行結果に示すように，1234Hのデータが8ビットの右ローテートによって，3412Hになったことがわかります．

リスト5．5●ror命令のプログラム例（xを8ビット右ローテートし，表示する）

```
 1: .model   small              ;メモリモデルsmall
 2: ;----------------------------------------------------
 3:   include iomac.inc         ;MS-DOSシステムコールのマクロ定義ファイル
 4:   extrn hexW :near          ;axの内容を16進表示する外部手続き
 5: ;-------------------------------
 6: .code ;コードセグメントの始まり
 7: start: mov ax,@data         ;データセグメントアドレス
 8:        mov ds,ax            ;データセグメントを設定
 9: ;
10:        mov cl,8             ;カウンタclに8を設定
11:        mov ax,x             ;変数xの値をaxへ
12:        ror ax,cl            ;axをclの値だけ、右ローテート
13:        mov y,ax             ;ローテート結果を変数yへ保存
14: ;
15:        mov ax,x             ;ローテート前の値をxへ移動
16:        call hexW            ;16進表示(axの内容)
17:        crlf                 ;改行
18:        mov ax,y             ;シフト結果yをaxへ移動
19:        call hexW            ;16進表示(axの内容)
20: ;
21:        exit                 ;MS-DOS復帰(マクロ)
22: ;-------------------------------
23: .data                       ;データセグメントの始まり
24: x      dw 1234H             ;変数x　初期値1234H
25: y      dw ?                 ;変数y　初期化せず
26: ;-------------------------------
27:        end start            ;startから実行開始
```

実行例

```
C:¥prg>ror
1234
3412
```

演習問題5

1. 次のビット操作命令を記述せよ．データセグメントとコードセグメントを設定する．
   (1) byte変数xをクリアする．
   (2) word変数wの値をビット反転する．
   (3) byte変数xの上位4ビットをマスクして，byte変数yへ保存する．
   (4) word変数xとyの論理和をとって，word変数zへ保存する．
   (5) word変数xを，1ビット右へ算術シフトする．
   (6) byte変数xを，4ビット左へシフトする．
   (7) byte変数xの無符号データを，乗算を用いずに8倍する．
   (8) word変数xの無符号データを，除算を用いずに1／4にする．
   (9) byte変数xの値を，byte変数yの値だけ左へローテートする．
   (10) word変数xをDIレジスタを用いた間接アドレスで，byte変数nの値だけ右へローテートする．

2. 次のプログラムを記述し，アセンブルして実行せよ．必要なら，6章で定義する2～16進表示サブルーチンを用いよ．
   (1) キーボードから数字を入力し，LSBをマスクして文字として表示する．
   (2) word変数xに適当な値を設定し，キーボードから入力した1桁の数だけ右または左シフトして，元データと結果を16進表示する．
   (3) word変数xに適当な値を設定し，キーボードから入力した1桁の数だけ右または左ローテートして，元データと結果を16進表示する．
   (4) 変数cに英小文字を初期値で与え，これを大文字に変換して表示する．
   ヒント　小文字のbit5をマスクする．
   (5) キーボードから英大文字を入力し，小文字に変換して表示する．

第6章

制御分岐命令

基本的にプログラムは若い番地の命令から順に実行されていきます．しかしながら，実際のプログラムでは，繰り返し同じ処理を実行したり，必要ない部分を飛ばしたりする場合が出てきます．特に，先に説明した演算命令の結果により制御を分ける条件判断は，プログラムの作成において重要な概念になります．本章では，まず条件判断に使用する無条件分岐命令，条件付分岐命令について説明します．

また，特定の処理をブロック化し，メインの処理からこのブロックを何度も呼び出すサブルーチンというしくみがあります．大きな処理をサブルーチンにブロック化することは，プログラムの点検・修正を小さなブロックで完結にすることが可能となるだけでなく，過去に作成したプログラムを繰り返し使用することも可能になります．本章の後半では，このサブルーチン呼び出しに使用するコール，リターン命令について説明します．

表6. 1に制御分岐命令を示します．表に示すように，無条件分岐命令，条件付分岐命令，ループ命令，手続き処理(サブルーチン呼び出し，復帰処理)命令，ソフトウェア割り込み命令があります．ここでは，ソフトウェア割り込み命令を除いた制御分岐命令についてとりあげます．

表6．1●制御分岐命令一覧

	命令	機能	
無条件分岐	JMP	jump	
条件付分岐	JA/JNBE	jump if above/not below nor equal	無符号データで、＞なら分岐
	JAE/JNB	jump if above or equal/not below	無符号データで、＞＝なら分岐
	JB/JNAE	jump if below/not above nor equal	無符号データで、＜なら分岐
	JBE/JNA	jump if below or equal/not above	無符号データで、＜＝なら分岐
	JC	jump if carry	桁上げがあれば分岐
	JE/JZ	jump if equal/zero	＝なら文岐
	JG/JNLE	jump if greater/not less nor equal	有符号データで、＞なら分岐
	JGE/JNL	jump if greater or equal/not less	有符号データで、＞＝なら分岐
	JL/JNGE	jump if less/not greater	有符号データで、＜なら分岐
	JLE/JNG	jump if lessor equal/not greater	有符号データで、＜＝なら分岐
	JNC	jump if not carry	桁上げがなければ分岐
	JNE/JNZ	jump if not equal/not zero	≠なら分岐
	JNO	jump if not overflow	オーバフローがなければ、分岐
	JNP/JPO	jump if not parity/parity odd	奇数パリティでなければ分岐
	JNS	jump if not sign	負でなければ分岐
	JO	jump if overflow	オーバフローならば、分岐
	JP/JPE	jump if parity/parity even	偶数パリティならば、分岐
	JS	jump if sign	負であれば分岐
ループ	JCXZ	jump if register CX=0	(CX)＝0ならばループ
	LOOP	loop	(CX)－1≠0ならばループ
	LOOPE/LOOPZ	loop if equal/zero	(ZF)=1であり、(CX)－1≠0ならばループ
	LOOPNE/LOOPNZ	loop if not equal/not zero	(ZF)=0であり、(CX)－1≠0ならばループ
手続き処理	CALL	call procedure	手続きの呼び出し
	RET	return from procedure	手続きからの復帰
割り込み	INT	interrupt	割り込み
	INTO	interrupt if overflow	オーバフロー割り込み
	IRET	interrupt return	割り込み処理からの復帰

6.1 無条件・条件付分岐命令

■(a)無条件分岐命令(相対ジャンプ)

プログラムはコンピュータのメモリ上に記憶され，アドレスの若い順に実行されていきます．ジャンプ命令はこの命令実行順序を変更する命令です．ジャンプ命令は次のようにラベルを付けられた番地を指定することで実行できます．

リスト6．1にjmp命令の使用例を示します．アセンブラでは，行き先の命令にラベルを付け，jmp命令側ではそのラベルを指定します．この例では行1:のadd命令の後，行4:のnextで示されるmov命令が実行されます．したがって，行3:のmov命令は実行されません．

リスト6．1●jmp命令の使用例

```
1:          add   bx,cx
2:          jmp   next            ;nextラベルへジャンプ
3:          mov   bx,0            ;この命令は実行されない
4: next:    mov   ax,bx           ;add命令の次に実行される
```

プロセッサ内部においてjmp命令は，インストラクションポインタ(IP)を変更することで実現されます．このIPの変更は指定された行先アドレスを直接代入するのではなく，行先アドレスと現在のIPが示すアドレスとの相対値を加算することにより実現されます．これは，通常のプログラムは近いアドレスへのジャンプが多いこと，プログラムをリロケータブル(再配置可能)にするためです．リロケータブルなコード(実行形式プログラム)が生成できることは，いろいろな利点があります．リロケータブルなプログラムはメモリ上のどこに配置しても実行可能になります．

Z80などのマイクロプロセッサのジャンプ命令では，実際のメモリ番地(絶対番地)を指定します．このため，Z80では指定したメモリ番地へプログラムを配置する必要があります．8086では相対番地を用いたジャンプ命令を用いることができるため，メモリ配置の自由度が大きいのです．

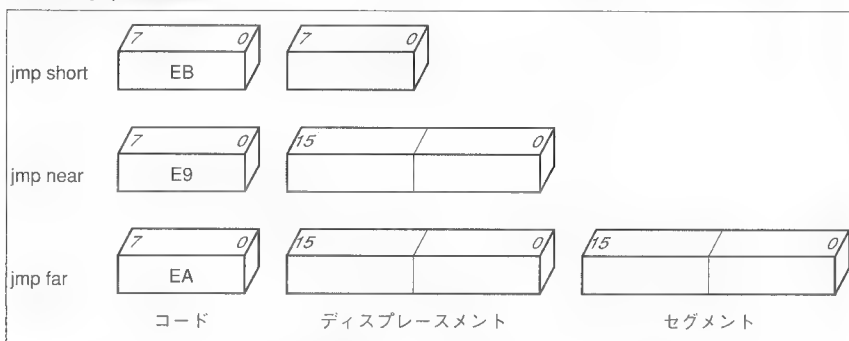
jmp命令のオペランドの長さは，行き先と現在のIPとの相対的な距離により以下のように3種類を選択できます．jmp命令の構成を**図6．1**に示します．

第6章 制御分岐命令

1 バイト (short jump) 現在の IP の示す値から－128～127 の範囲にジャンプする場合

2 バイト (near jump) 現在の IP の示す値から－32768～32767 の範囲にジャンプする場合

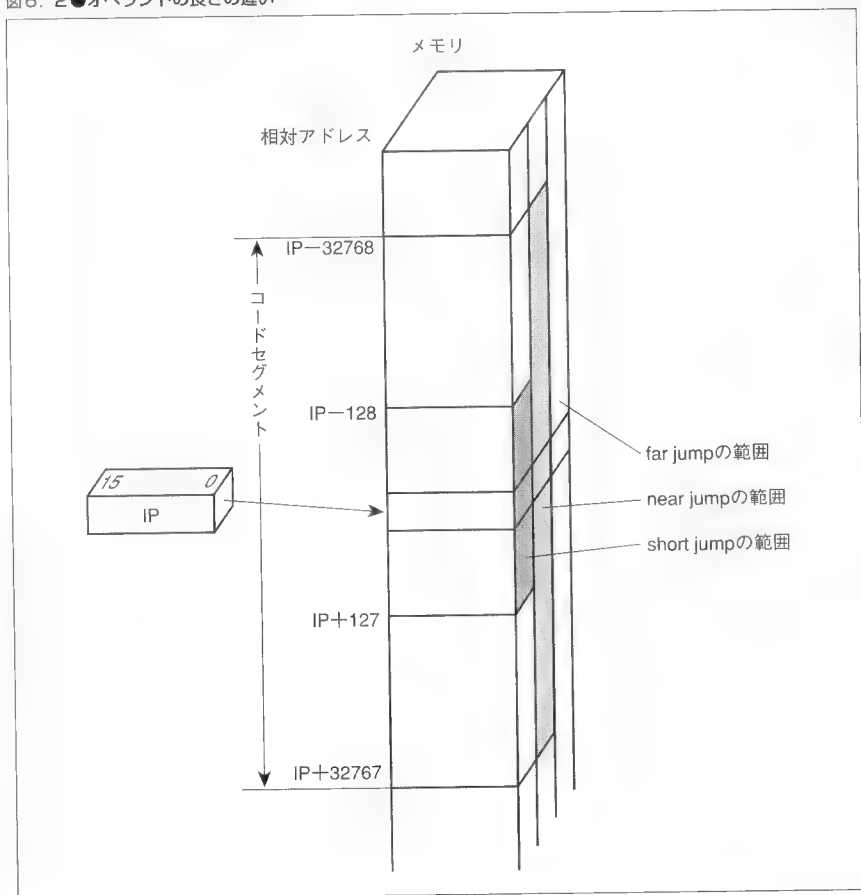
4 バイト(far jump) 2バイトでは届かない範囲にジャンプする場合

図6．1●jmp命令の構成

当然ながらオペランドが短いほど，命令をフェッチする際にメモリアクセスの回数が減り，命令実行時間が短くなります．したがって，行先アドレスとの距離の短い場合には，なるべくバイト数の少ない命令を使用することが望まれます．これらを明示的に指示するためには，ラベルの前にそれぞれ

- **short**
- **near ptr**
- **far ptr**

を記述します．これらを指示しなかった場合のオペランドの長さは，使用するアセンブラにより異なります．jmp 命令より後方にあるラベル(後方参照　アドレスの低い方向)へのジャンプの場合，行先アドレスとの距離が既知であるため，多くのアセンブラでは最適なオペランドの長さを自動的に設定してくれます．jmp 命令の前方にあるラベル(前方参照　アドレスの高い方向)へのジャンプでは，行き先アドレスとの距離が未知であるため，最適なオペランドの長さを設定することはできません．指定がない場合には，アセンブラはオペランドの長さを大きめに設定します．もし，行先アドレスまでの距離が短い場合には，余

った部分にnop命令が挿入される場合があります．

図6．2●オペランドの長さの違い

このようにオペランドに，行先までのアドレスの距離を指定するものを相対ジャンプといいます．この相対値がどのように計算されるかをみてみましょう．

リスト6．2はジャンプ命令とそのコードを示しています．まず行4:のjmp命令を見てみます．この命令では特にオペランドの長さを指定していませんが，アセンブラにより自動的にshort jmp (2バイト命令)が選択されています．オペランドの値0AHは，現在のIPの値 (0002H) と行き先ラベル l1 が示すアドレス(000CH)の相対値を示しています．ここでjmp命令のフェッチはすでに終わっ

ているため，IPの値は次の命令を示していることに注意してください．near ptr jmpの場合には，オペランドの長さが2バイトになっている以外は，short jmpと同じです．こちらは3バイト命令になります．

一方，far ptr jmp の場合には，オペランドにアドレスの情報と同時にセグメントの情報がオペランドに入ります．セグメント情報はリンカによって割り付けられるため，ここでは ---- と示されています．また，アドレスの隣の'R'は実際には相対アドレスとして後で計算されることを示します．このように，far ptr jmp はセグメント情報も含めたジャンプであるため，セグメントを超えた距離の離れたジャンプが行えます．

また，後方ジャンプの場合には，相対値は負の値になります．short jmpの場合には，16ビットに符号拡張されて足し算されます．たとえば，行10:のjmp命令ではオペランドが0FDhであるので，符号拡張され 0FFFDhとなります．したがって，

000Fh (現在の IP) + 0FFFDh (オペランド) = 000Ch

という計算で行き先アドレスが計算されます．

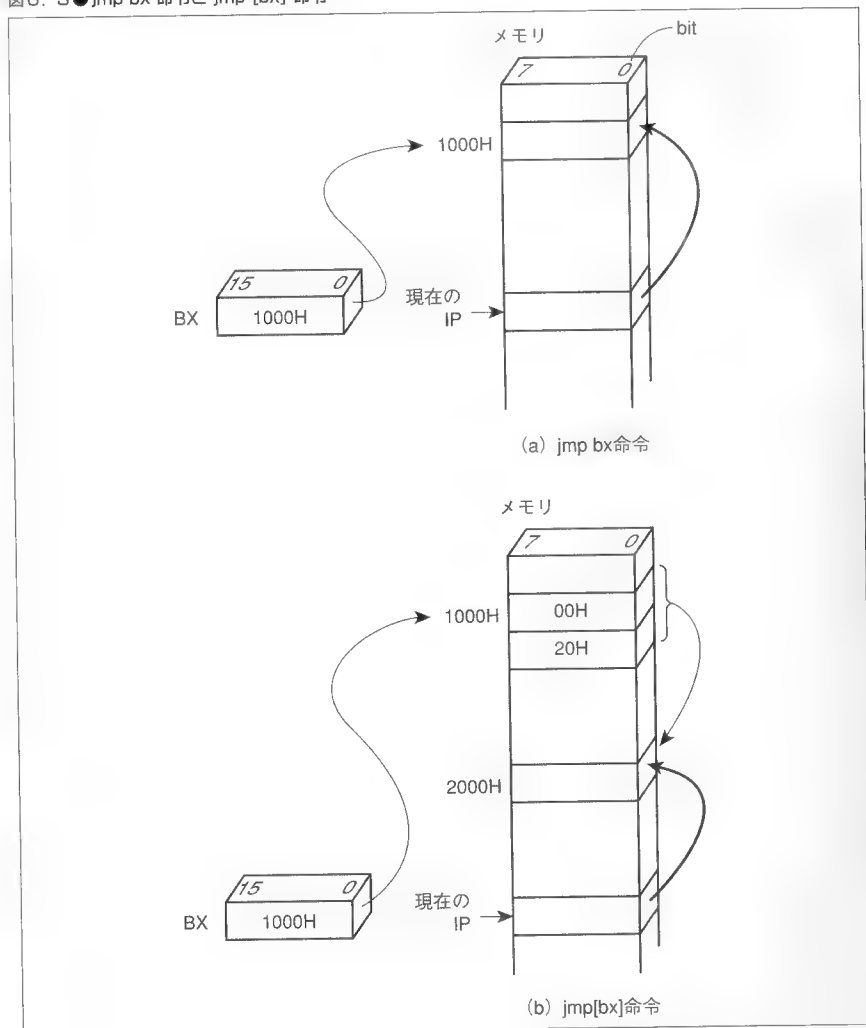
リスト6．2●アドレス計算

```
 1:                          .model      small
 2: 0000                     .code
 3: 0000              main   proc far
 4: 0000  EB 0A       start: jmp  l1            ;順方向 short jmp
 5: 0002  E9 0007            jmp  near ptr l1   ;順方向 near jmp
 6: 0005  EA ---- 000C R     jmp  far  ptr l1   ;順方向 far jmp
 7: 000A  90                 nop                ;何の働きもしない命令
 8: 000B  90                 nop
 9: 000C  90          l1:    nop
10: 000D  EB FD              jmp  l1            ;逆方向 short jmp
11: 000F  E9 FFFA     jmp    near ptr l1        ;逆方向 near jmp
12: 0012  EA ---- 000C R jmp far  ptr l1        ;逆方向 far jmp
13: 0017              main   endp
14:                          end         start
```

ここまで示したジャンプ命令はすべて行き先が固定なものでしたが，場合によっては行き先を可変にする必要があります．このような場合には，jmp bxや

jmp [bx]という命令が使えます．前者は，BXレジスタの内容を行き先のアドレスとみなし，そのアドレスにジャンプします．後者はBXレジスタの値が示すアドレスのメモリに行き先のアドレスが書かれているとみなし，そのアドレスにジャンプします．それぞれを，直接アドレス指定，間接アドレス指定と呼びます．

図6．3●jmp bx 命令と jmp [bx] 命令

第6章 制御分岐命令

BXレジスタを利用したジャンプの例として**リスト6．3**にテーブルジャンプのプログラムを示します．このプログラムはキーボードから入力された数字が4で割り切れるか，偶数であるか，奇数であるかを表示するプログラムです．

リスト6．3●テーブルジャンプのプログラム例

```
 1: .model    small
 2:           include  iomac.asm  ;マクロ定義ファイルの読み込み
 3: ;---------------------------------------------
 4: .code                         ;コードセグメント
 5: tbljmp    proc   far
 6:           mov    ax,@data     ;データセグメントを設定
 7:           mov    ds,ax
 8: ;
 9: keyin                         ;1 文字入力
10:           sub  al,'0'         ;'0'を引いて数値に変換
11:           and    ax,3         ;axを0003Hでマスクを
12:           lea    bx,jtable    ;テーブルの先頭アドレス
13:           add    ax,ax        ;axの値を2倍に
14:           add    bx,ax        ;対応する番号のテーブルアドレスを計算
15:           jmp    cs:[bx]      ;テーブルに書かれたアドレスにジャンプ
16: jdiv4:    lea    dx,sdiv4     ;テーブルアドレス
17:           disps               ;4で割れることを表示
18: jeven:    lea    dx,seven     ;テーブルアドレス
19:           disps               ;偶数であることを表示
20:           exit                ;終了
21: jodd:     lea    dx,sodd      ;テーブルアドレス
22:           disps               ;奇数であることを表示
23:           exit                ;終了
24: tbljmp    endp
25: ;---------------------------- ;ジャンプアドレス
26: jtable:   dw     jdiv4        ;4で割り切れるとき
27:           dw     jodd         ;4で割って1余るとき
28:           dw     jeven        ;4で割って2余るとき
29:           dw     jodd         ;4で割って3余るとき
30: ;---------------------------------------------
31: .data                         ;データセグメント
32: sdiv4     db     'この数字は4で割り切れます．', CR, LF, '$'
33: seven     db     'この数字は偶数です．', CR, LF, '$'
34: sodd      db     'この数字は奇数です．', CR, LF, '$'
35: ;---------------------------------------------
36: .stack    100h                ;スタック領域100h
37:           end    tbljmp
```

```
実行例
    C:prg>jmp
    4この数字は4で割り切れます.
    この数字は偶数です.
```

■(b)条件付分岐命令

プログラムの流れを演算結果に従って変更する命令が条件分岐命令です．一般的に

jcc　　label　　　　;ccは条件により変わる

のように書きます．条件分岐命令は2バイトの命令でshort jmpのみが許されます．また，ccはcondition codeの略で分岐のための条件を指定します．たとえば，条件分岐の前に行った演算結果が0になった場合，フラグZFがセット(1)されます．したがって，この演算命令の後に，

jz　　label

という命令があった場合には，labelのアドレスにジャンプが行われます．逆に，演算結果が0でなかった場合には，この命令は無視され次の命令が実行されます．

ccには，各フラグレジスタの状態をテストできるように以下のものが用意されています．

z　　ゼロになった場合
nz　　ゼロでなかった場合
c　　キャリが生じた場合
nc　　キャリが生じなかった場合
s　　結果が負の場合
ns　　結果が正の場合
o　　オーバフローがあった場合
no　　オーバフローがなかった場合
p　　パリティがセット(偶数)の場合
np　　パリティがリセット(奇数)の場合

また，条件分岐の前の命令が比較命令の場合には同時に複数のフラグをテストしないと大小関係が比較できません．そのため，次のようなccも用意されています．ここで，条件分岐命令の前に符号付きの演算が行われたか，符号なし

の演算が行われたかによって次のように異なることに注意してください.

■符号付演算後の Condition Code

g	**A > B (Greater)**
ge	**A ≧ B (Greater or Equal)**
e	**A = B (Equal)**
ne	**A ≠ B (Not Equal)**
l	**A < B (Less)**
le	**A ≦ B (Less or Equal)**

■符号なし演算後の Condition Code

a	**A > B (Above)**
ae	**A ≧ B (Above or Equal)**
e	**A = B (Equal)**
ne	**A ≠ B (Not Equal)**
b	**A < B (Below)**
be	**A ≦ B (Below or Equal)**

このように条件分岐命令ではフラグの状態が非常に重要になります.したがって,命令を実行したときにどのフラグレジスタが影響を受けるのかを知っておく必要があります.詳しくは付録の命令表を参照してください.

■(c)ループ命令

プログラムでは同じ処理の繰り返しが使われることが多いため,8086にはこの繰り返しを行うためのloop命令が用意されています.loop命令は条件分岐命令と同様に1バイトの相対値をオペランドに持つジャンプ命令です.繰り返しという仕組みから,基本的には後方へのジャンプとなる場合がほとんどです.書式は,

```
loop      label                ;無条件ループ
loopcc    label                ;cc は jcc と同じ condition code
```

となります．

loop命令では，CXレジスタをカウンタとみなし，命令を実行するたびにCXの値が1つずつ減じられます．この減算後にCXの値が0になった場合には，指定された条件に関わらずジャンプは行われません．loopを用いた例として，1から10までの足し算を計算するプログラムを，リスト6．4に示します．このプログラムではloop命令をうまく使うために，10から1まで逆順に足しています．

リスト6．4●loop命令（1から10までの加算）

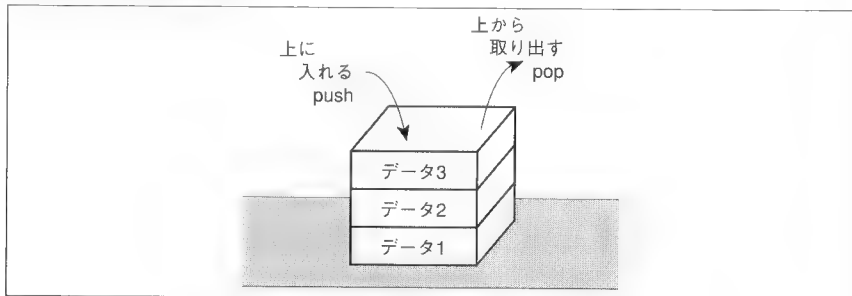
```
;

 1: .model small
 2: ;------------------------------------
 3:         extrn hexW:near
 4:         include          iomac.inc
 5: ;------------------------------------
 6: .code                    ;コードセグメント
 7: start:
 8:         mov ax,@data     ;@dataはセグメントアドレス
 9:         mov ds,ax        ;DSへデータセグメントの設定
10: ;-----------------------
11:         mov ax,0         ; 結果を入れるレジスタの初期化
12:         mov cx,10        ; 最終値の設定
13: loop1:  add ax,cx        ; cx の値を足していく
14:         loop loop1       ; cx が 0 になる
15:         call hexW        ; ax レジスタを表示
16:         crlf             ; 改行を表示(マクロ)
17:         exit             ; MS-DOS へ復帰
18: ;-------------------------------------------
19: .stack 100H              ;スタックセグメント
20: ;-------------------------------------------
21:         end                   start

実行例
    C:¥prg>loop
    0037
```

第6章 制御分岐命令

6.2 スタック

サブルーチンを呼び出す際に，スタックが重要な働きをします．スタックについては，3章でも取り上げましたが，重要な概念ですからもう一度説明します．

データなどを一時的に保存しておくためのスペースとして，アセンブラではスタックという特殊なメモリ空間を使用します．スタックへのデータの保存は**図6．4**に示すように，机の上に本を置いておくのと同じ要領で，記憶させるデータは過去に記憶されたデータの上に置いていきます．逆にデータを取り出す場合には，上に置いたデータから順番に取り出していきます．この仕組みのことをFILO (first in-last out：先入後出)またはLIFO(last in-first out：後入先出)といいます．8086では，FILO用のメモリ空間としてスタックセグメントSSが用意され，そのアクセスのために専用のアドレスレジスタSPが使用されます．

図6．4●スタックの概念

レジスタのデータをスタックに積む命令としてpush命令が，逆にスタックからデータを取り出す命令としてpop命令が用意されています．たとえば，push axという命令は，

- **SPの値を2減らす**
- **SS:SPのアドレスにaxレジスタの内容を格納する**

という2つの作業を行います．逆にpop axという命令では，

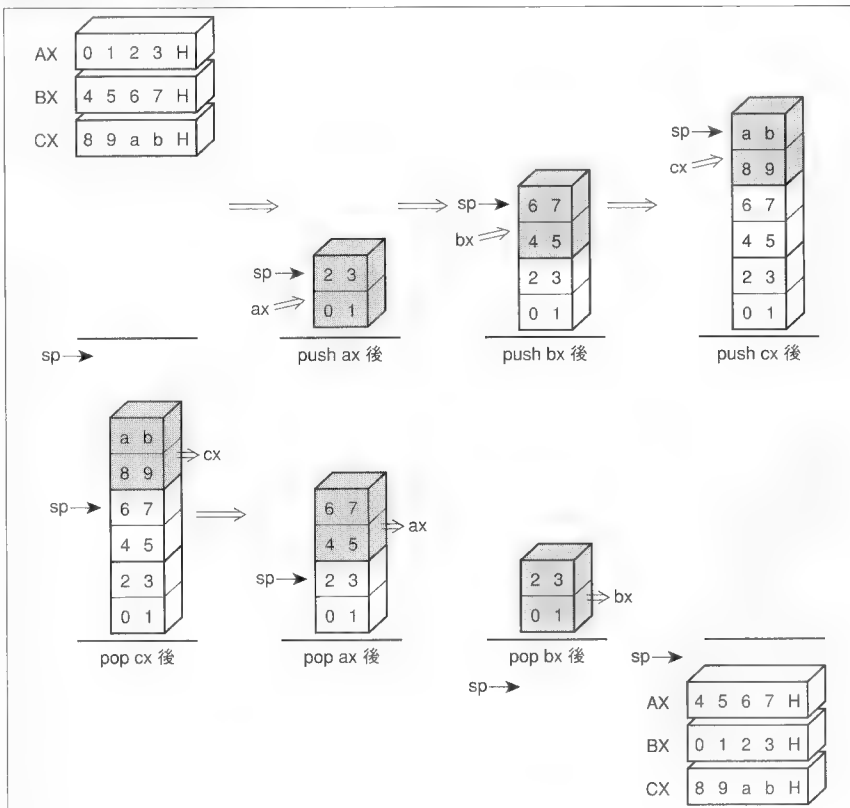
- **SS:SPのアドレスからデータを読み出し，axに格納する**
- **SPの値を2増やす**

という2つの作業を行います．例として，**リスト6．5**を実行したときのスタッ

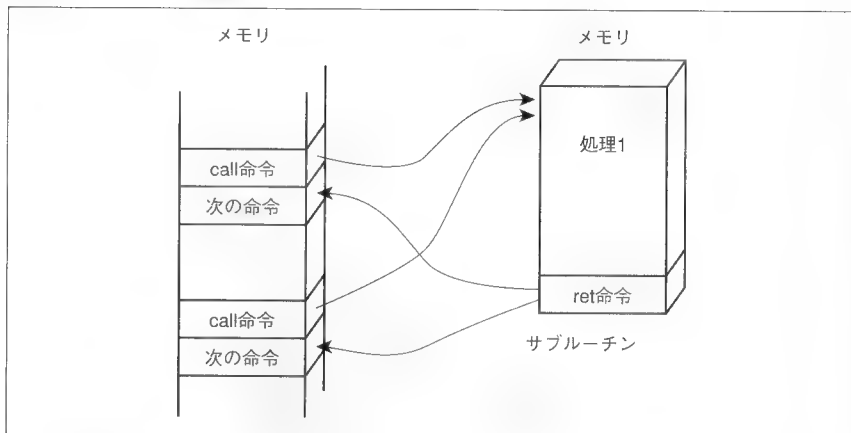
クの変化の様子を図6．5に示します．このようにスタックを使うと，レジスタのデータ退避以外にも，レジスタ同士のデータの入れ換えも行うことができることがわかります．

リスト6．5●push命令，pop命令

```
1: .code
2: push  ax
3: push  bx
4: push  cx
5: mov   cx,1
6: pop   cx
7: pop   ax
8: pop   bx
```

図6．5●スタックの変化の様子

6.3 コールリターン命令

プログラムを作成していると，同じ処理を何度も実行することが多いことに気づきます．このような場合には，**図6．6**に示すように個々の処理を1つのパッケージとして完結させます．それらのパッケージを呼び出すという手順を取ると，プログラムの作成が効率的になります．アセンブラではこのようなパッケージのことを，サブルーチンやプロシージャ(手続き)などと呼びます．このサブルーチン呼び出しに用いられるのがcall命令とret命令です．

図6．6●サブルーチンの概念

リスト6．6にcall命令の使用例を示します．この例ではnearプロシージャlnearとfarプロシージャlfarを呼び出しています．call命令とjump命令は，short callがない以外は同じオペランド構成をとります．jmp命令との大きな違いは，call命令では一連の処理が修了した後に，ret命令によりcall命令の次の命令に戻れることです．この元に戻るという処理を実現するために，call命令では現在のIPの値を先ほど説明したスタックに積むという作業を行っています．

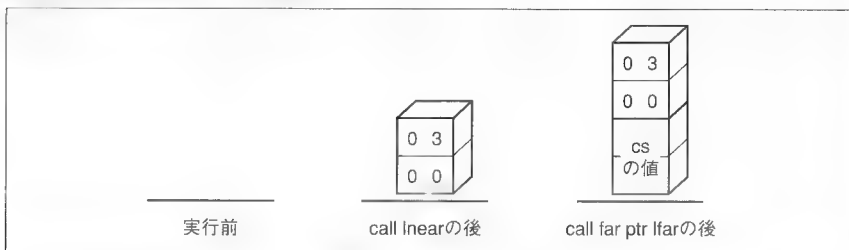
ret命令ではこのスタックから戻るべきアドレスを取り出し，IPにその値を戻すことにより，サブルーチンから元の処理に戻る作業を行います．

リスト6.6●コール命令の例

```
                          .model small
0000                      .code
0000                      excall proc   far
0000 E8 000A                     call   lnear        ; near call
0003 9A ---- 000E R              call   far ptr lfar ; far call
0008 0E E8 0002                  call   lfar         ; near call
000A 0E                          push   cs           ; 参考
000D                      excall endp
000D                      lnear  proc                ; near procedure
000D C3                          ret
000E                      lnear  endp
000E                      lfar   proc                ; far procedure
000E CB                          ret
000F                      lfar   endp
                                 end    excall
```


ここで，lnearプロシージャとlfarプロシージャのret命令のコードを見てみましょう．同じretという命令であるにも関わらず，lnearではC3H，lfarではCBHという異なるコードが出力されていることがわかります．これは，**図6.7**に示すようにnear callとfar callではスタックに積む情報が異なるためです．near callでは近いアドレスからのcallであるために，スタックにはIPの値のみが格納されるのに対し，far callではIPの値と同時に呼び出し元のコードセグメントの情報も格納されます．このことから，call命令はプロシージャの型に合わせて呼び出すべきであることがわかります．

もし，farプロシージャをnear callしてしまった場合にはどうなるでしょうか．farプロシージャのret命令はスタックにIPとコードセグメントの情報があることを期待していますが，near callではコードセグメントの情報は積まれません．この不都合を回避するために，アセンブラは自動的にnear callの前にコードセグメントの値をスタックに積む動作を行います．**リスト6.6**の0008番地の命令に対応するコードを見るとわかるように，near callのコードの前に000A番地のpush csと同じコードが挿入されていることがわかります．実際にはこのようにアセンブラが補助してくれますが，混乱を避けるためにも，farプロシージャはfar callするようにするべきです．

図6．7●call命令後のスタックの状態

6.4 基本サブルーチン

ここではサブルーチンの実際例として，MS-DOSシステムコールを機能ごとにサブルーチン化します．基本サブルーチンは，簡略化セグメント定義と完全なセグメント定義の両方を示します．

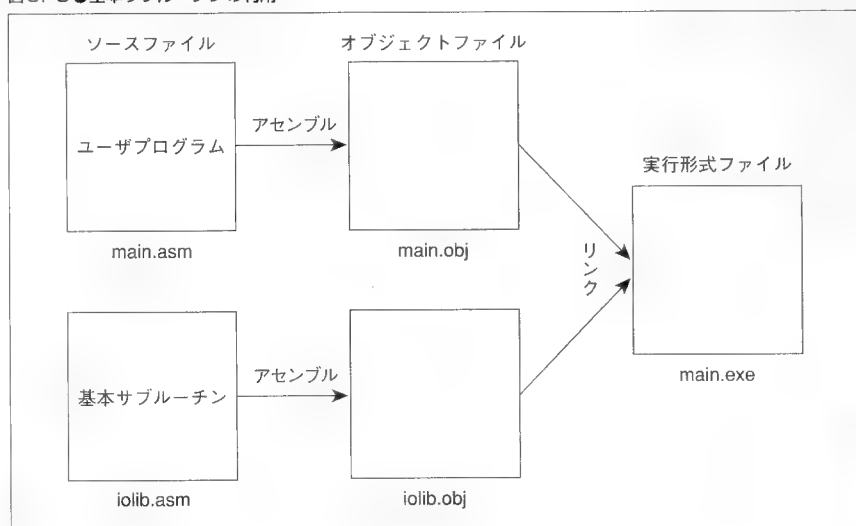
この章で用いる簡略化セグメント定義の基本サブルーチンを**リスト6．7**に示します．このサブルーチン群はMS-DOSを用いた入出力プログラムを定義しています．**図6．8**に示すように，これら基本サブルーチンは，ユーザのメインプログラム(図ではmain.asm)とは別のファイル(iolib.asm)に保存し，別にアセンブルしてオブジェクトファイル(iolib.obj)として保存します．リンク時にメインプログラムとこの基本サブルーチンとを結合して，実行形式のプログラム(main.exe)を作成します．MASMのライブラリ機能とは異なりますが，一種のライブラリと考えてもよいでしょう．1度アセンブルしてオブジェクトファイルを作成しておけば，どのユーザプログラムからもリンクして使用できるようになります．このような利用方法を用いることで，プログラムのバグの発生を少なくすることができます．

これら2つのプログラムのアセンブルとリンク方法は次の通りです．

```
C:¥prg>ml /c /Zm /Fl /Fm main.asm iolib.asm
C:¥prg>link main iolib
```

ここで，/cオプションを削除して次のようにすると，アセンブルとリンクを同時に行うことができます．

```
C:¥prg>ml /Zm /Fl /Fm main.asm iolib.asm
```

図6．8●基本サブルーチンの利用

基本サブルーチンの機能を表6．2に示します．これらサブルーチンはすべてMS-DOSシステムコールを用いて実現しています．keyinはMS-DOSファンクション1を用いて，キーボードから1文字入力を行います．入力1文字はALレジスタで戻されます．このルーチンでの使用レジスタはAHとALレジスタで，これらレジスタの元の値は変更される可能性があります．またMS-DOSファンクション内でレジスタが使用される可能性があるため，安全のためには全レジスタの保存と回復のプログラムを付けるとよいでしょう．

keyin0はエコーバックなしにキーボードから1文字入力します．入力された文字は，ALレジスタを用いて戻します．keyinsは文字列を出力するもので，文字列を保存するバッファアドレスを，DXレジスタに設定してからkeyinsを呼び出します．

dispはディスプレイへ1文字表示するもので，表示文字をDLレジスタへ設定してから，このdispを呼び出します．crlfはディスプレイのカーソルを現在ある位置から，次の行の先頭へ移すためで，補助的に使用します．printはプリンタへ1文字出力するもので，出力文字をDLレジスタへ設定してから，printを呼び出します．

exitはプロセスを終了して，このプロセスを呼び出した親プロセスへ復帰す

るものです．実際にはユーザプログラムを終了して，MS-DOSへ制御を戻します．そのためサブルーチンが呼び出されたプログラムへ復帰するためのret命令がありません．

表6．2●基本サブルーチン一覧

サブルーチン名	機能	呼び出し	戻り値	使用レジスタ
keyin	キーボードから１文字入力（エコーバックあり）		AL　入力文字	AH,AL
keyin0	キーボードから１文字入力（エコーバックなし）		AL　入力文字	AH,AL
keyins	キーボードから文字列を入力	DX←バッファの先頭アドレス		AH,DX
disp	ディスプレイへ１文字表示	DL←表示文字		AH,DL
crlf	ディスプレイを改行			AH,DL
disps	ディスプレイへ文字列を表示	DX←文字列の先頭アドレス		AH,DX
print	プリンタへ1文字出力	DL←出力文字		AH,DL
exit	プロセスの終了 MS-DOSへ復帰			AH,AL

リスト6．7のプログラムは，簡略化セグメント疑似命令を用いて簡略化したセグメント定義を行っています．このサブルーチンを用いるプログラムも簡略化セグメント定義をする必要があります．プログラムでは最初にモデルをsmallとして宣言し，次にサブルーチン群が外部から参照できるようにpublic宣言をしています．コードセグメントは.codeとして領域を定めています．

MS-DOSファンクション8を指定するためAHレジスタにこれを与えています．ファンクション8はALレジスタへキーボードから1文字をエコーなしで入力する関数です．エコーとは，入力した文字をディスプレイ上に表示することをいいます．ファンクション8はエコーなしのキー入力ですから，ディスプレイには何も表示されません．実際のシステムコールは行10:のint 21hのソフトウェア割り込み命令で生じます．

リスト6．7●基本サブルーチン（簡略化セグメント定義）iolib.asm

```
 1: ;
 2: ;   基本サブルーチン(簡略化セグメント定義)
 3: ;
 4: .model  small
 5:     public keyin,keyin0,keyins,disp,disps,crlf,print,exit,crlf
                                ;外部から呼び出し可能
 6: .code
 7: ;--------------------------------
 8: keyin proc near             ;キーボードから1文字入力(エコーバック). ALに入力文字
 9:     mov ah,1h               ;ファンクション1を指定
10:     int 21h                 ;MS-DOSのシステムコール(割込み)
11:     ret                     ;復帰
12: keyin endp
13: ;--------------------------------
14: keyin0 proc near            ;キーボードから1文字入力(エコーバックなし). ALに入力文字
15:     mov ah,8h               ;ファンクション8を指定
16:     int 21h                 ;MS-DOSのシステムコール(割込み)
17:     ret                     ;復帰
18: keyin0 endp
19: ;--------------------------------
20: keyins proc near            ;キーボードから文字列をバッファへ入力. DXにバッファアドレス
21:     mov ah,0ah              ;ファンクション0a指定
22:     int 21h                 ;MS-DOSのシステムコール(割込み)
23:     ret                     ;復帰
24: keyins endp
25: ;--------------------------------
26: disp  proc near             ;ディスプレイに1文字出力. DLに出力文字
27:     mov ah,2h               ;ファンクション2を指定
28:     int 21h                 ;MS-DOSのシステムコール(割込み)
29:     ret                     ;復帰
30: disp endp
31: ;--------------------------------
32: disps proc near             ;ディスプレイに文字列を表示. DXに文字列アドレス
33:     mov ah,9h               ;ファンクション9を指定
34:     int 21h                 ;MS-DOSのシステムコール(割込み)
35:     ret                     ;復帰
36: disps endp
37: ;--------------------------------
38: print proc near             ;プリンタから1文字印刷. DLに印刷文字
39:     mov ah,5h               ;ファンクション5を指定
40:     int 21h                 ;MS-DOSのシステムコール(割込み)
41:     ret                     ;復帰
42: print endp
```

```
43: ;------------------------------
44: exit proc near              ;プロセスの終了して，MS-DOSへ復帰
45:      mov ax,4c00h           ;ファンクション4cを指定．リターンコード0
46:      int 21h                ;MS-DOSのシステムコール(割込み)
47: exit endp
48: ;------------------------------
49: crlf  proc near             ;ディスプレイを1行改行
50:      mov dl,0ah             ;CR(復帰)コード
51:      call disp              ;表示サブルーチンをコール
52:      mov dl,0dh             ;LF(改行)コード
53:      call disp              ;表示サブルーチンをコール
54:      ret                    ;復帰
55: crlf endp
56: ;------------------------------
57:      end
```

リスト6．8にまったく同じ働きの，完全なセグメント定義をした基本サブルーチンを示します．一般のセグメント疑似命令を用いたユーザプログラムでは，こちらの基本サブルーチンを用います．ここでも各サブルーチンが外部から参照できるようにpublic宣言をしています．

MS-DOSに依存しないプログラムを作成する場合は，これらサブルーチンをコンピュータに合わせて記述し直す必要があります．以下に，これらサブルーチン群を用いた基本プログラムを示します．

リスト6．8●基本サブルーチン(完全なセグメント定義)

```
 1: ;
 2: ;基本サブルーチン(完全なセグメント定義)
 3: ;
 4:  assume cs:mycode
 5:  public keyin,keyin0,keyins,disp,disps,crlf,print,exit,crlf
                                ;外部から呼び出し可能
 6: ;
 7: mycode segment public
 8: ;------------------------------
 9: keyin proc near             ;キーボードから1文字入力(エコーバック)．ALに入力文字
10:      mov ah,1h              ;ファンクション1を指定
11:      int 21h                ;MS-DOSのシステムコール(割込み)
12:      ret                    ;復帰
```

```
13: keyin  endp
14: ;-------------------------------
15: keyin0 proc near            ;キーボードから1文字入力(エコーバックなし). ALに入力文字
16:     mov ah,8h               ;ファンクション8を指定
17:     int 21h                 ;MS-DOSのシステムコール(割込み)
18: ret                         ;復帰
19: keyin0 endp
20: ;-------------------------------
21: keyins proc near            ;キーボードから文字列をバッファへ入力. DXにバッファアドレス
22:     mov ah,0ah              ;ファンクション0a指定
23:     int 21h                 ;MS-DOSのシステムコール(割込み)
24:     ret                     ;復帰
25: keyins endp
26: ;-------------------------------
27: disp  proc near             ;ディスプレイに1文字出力. DLに出力文字
28:     mov ah,2h               ;ファンクション2を指定
29:     int 21h                 ;MS-DOSのシステムコール(割込み)
30:     ret                     ;復帰
31: disp  endp
32: ;-------------------------------
33: disps proc near             ;ディスプレイに文字列を表示. DXに文字列アドレス
34:     mov ah,9h               ;ファンクション9を指定
35:     int 21h                 ;MS-DOSのシステムコール(割込み)
36:     ret                     ;復帰
37: disps endp
38: ;-------------------------------
39: print proc near             ;プリンタから1文字印刷. DLに印刷文字
40:     mov ah,5h               ;ファンクション5を指定
41:     int 21h                 ;MS-DOSのシステムコール(割込み)
42:     ret                     ;復帰
43: print endp
44: ;-------------------------------
45:     exit proc near          ;プロセスの終了して, MS-DOSへ復帰
46:     mov ax,4c00h            ;ファンクション4cを指定.リターンコード0
47:     int 21h                 ;MS-DOSのシステムコール(割込み)
48: exit endp
49: ;-------------------------------
50: crlf  proc near             ;ディスプレイを1行改行
51:     mov dl,0ah              ;CR(復帰)コード
52:     call disp               ;表示サブルーチンをコール
53:     mov dl,0dh              ;LF(改行)コード
54:     call disp               ;表示サブルーチンをコール
55:     ret                     ;復帰
56: crlf  endp
```

```
57: ;------------------------------
58: mycode   ends
59:      end
```

6.5 2進，10進，16進表示サブルーチン

ここではサブルーチンの実際例として，2進コードを2進，10進，16進数として表示するサブルーチンを下に示します．これらのサブルーチンは，4章以降で実際に利用しています．

サブルーチン名	機能
hexB	**ALの内容を2桁の16進数で表示**
hexW	**AXの内容を4桁の16進数で表示**

リスト6．9に示す16進表示サブルーチンhexWについて簡単に説明します．hexWはAXレジスタにある2進数を，16進4桁で表示します．これを利用するプログラムは，次のような形式で利用することができます．

```
extrn hexW      ;外部手続きであることの宣言
mov ax,z        ;AXに2進数を代入．変数zにあるとする
call hexW       ;実際の手続き呼び出し
```

なお，このファイルにはALにある2進数を，16進2桁で表示するサブルーチンhexBも記述してあります．

numlib.asmをどのように，これを利用するプログラムに連結するかを説明します．今メインプログラムがmul.asmファイルに，hexWがnumlib.asmファイルにあるとすると，それぞれ別にアセンブルを行います．これで，mul.objとnumlib.objファイルが作られます．このオブジェクトファイル2つを次のようにリンクすると，mul.exeの実行形式プログラムを作ることができます．

C:¥prg>link mul numlib　　　**mul.objとnumlib.objのリンク → mul.exe**

C:¥prg>mul　　　**mul.exeの実行**

リスト6．9●16進表示サブルーチン（numlib.asm）

```
 1: .model      small
 2: ;
 3:   public    hexB,hexW     ;外部参照可
 4: ;------------------------------
 5: disp macro                  ;ディスプレイに1文字出力．DLに出力文字
 6:      mov ah,2h              ;ファンクション2を指定
 7:      int 21h                ;MS-DOSのシステムコール(割込み)
 8:      endm
 9: ;
10: .code
11: ;-----------------------------------------------------
12: ;  hexwd: ax の値を16進数4桁表示
13: ;
14: hexW proc   near
15:      xchg   ah,al           ;ahとalを交換
16:      call   hexB            ;もともとのaxの上位バイトを表示
17:      xchg   ah,al           ;ahとalを交換
18:      call   hexB            ;もともとのaxの下位バイトを表示
19:      ret
20: hexW endp
21: ;------------------------------
22: ;  hexbt: al の値を16進数2桁表示
23: ;
24: hexB proc   near
25:      push   ax              ;axが壊れるので退避しておく
26:      push   bx              ;bxが壊れるので退避しておく
27:      push   cx              ;cxが壊れるので退避しておく
28:      lea    bx,htable       ;16進数のキャラクタテーブルの先頭アドレスをセット
29:      mov    ch,al           ;alの内容をchに保存
30:      mov    cl,4            ;シフトするビット数(4)をclにセット
31:      shr    al,cl           ;上位4ビットを表示するために4ビット右シフト
32:      xlatb                  ;テーブルから文字を抜き出す
33:      mov dl,al              ;表示のため
34:      disp                   ;得られた文字を表示
35:      mov    al,ch           ;下位4ビットを表示するためalを復帰
36:      and    al,0fH          ;下位4ビットをマスク
37:      xlatb                  ;テーブルから文字を抜き出す
37:      mov dl,al              ;表示のため
38:      disp                   ;得られた文字を表示
```

第6章 制御分岐命令

```
39:       pop    cx              ;退避させておいたcxを復帰
40:       pop    bx              ;退避させておいたbxを復帰
41:       pop    ax              ;退避させておいたaxを復帰
42:       ret
43: hexB  endp
44: ;------------------------------------------
45:   .data
46:   htable    db  '0123456789ABCDEF'    ;数→文字変換テーブル
47:   ;----------------------------
48:             end
```

演習などの際に純2進数を2進，10進数で表示するサブルーチンを**リスト6. 10**に示します．このプログラムを**リスト6. 9**のnumlib.asmに追加すれば，バイトまたはワードデータを2進，10進，16進で表示できます．なお，追加されるサブルーチンは次の通りです．

サブルーチン名	機能
decB	**ALの内容を5桁の10進数で表示**
decW	**AXの内容を5桁の10進数で表示**
binB	**ALの内容を8桁の2進数で表示**
binW	**AXの内容を16桁の2進数で表示**

リスト6. 10●2進，10進表示サブルーチン　numlib.asmへの追加

```
     public binB,binW,decB,decW        ;外部参照可
;------------------------------
.code
;----------------------------------------------
;   decW: ax の値を 10 進数 5 桁表示
;
decW   proc   near
       push   ax            ;axが壊れるので退避しておく
       push   bx            ;bxが壊れるので退避しておく
       push   cx            ;cxが壊れるので退避しておく
;
;2進コードをBCD桁に分解
       mov cx,5             ;BCD分解のため、カウンタに桁数を設定
```

```
        lea di,bcd+5-1          ;BCDの保存アドレス
col:    xor dx,dx               ;dxをクリア
        div n10w                ;(dx:ax)/10 余りはBCD下位桁
        mov [di],dl             ;余りをBCDとして保存
        dec di                  ;BCD保存アドレスを1減. 上位桁へ
        loop col                ;cxを1減. 0でなければcolに飛び繰り返す
;
;BCD桁を10進表示
        mov cx,5                ;BCD桁数をカウンタに設定
        lea si,bcd              ;BCDの保存アドレスを設定
dspdw:  mov dl,[si]             ;BCD1桁をDLにロード
        add dl,'0'              ;BCDの値に数字0を加算→数字化
        disp                    ;BCDの1桁を10進表示
        inc si                  ;BCD保存アドレスを1増
        loop dspdw              ;cxを1減.0でなければdspに飛び繰り返す
        pop  cx                 ;退避させておいたcxを復帰
        pop  bx                 ;退避させておいたbxを復帰
        pop  ax                 ;退避させておいたaxを復帰
        ret
decW    endp
;-------------------------------------------
;   decB: al の値を 10 進数 5 桁表示
;
decB    proc   near
        push   ax               ;axが壊れるので退避しておく
        xor ah,ah
        call decW
        pop    ax               ;退避させておいたaxを復帰
        ret
decB    endp
;--------------------------------------------------
;   binW: ax の値を 2 進数 16 桁表示
;
binW    proc   near
        xchg   ah,al            ;ahとalを交換
        call   binB             ;もともとのaxの上位バイトを表示
        xchg   ah,al            ;ahとalを交換
        call   binB             ;もともとのaxの下位バイトを表示
        ret
binW    endp
;-------------------------------------------
;   binB: al の値を 2 進数 8 桁表示
;
binB    proc   near
```

```
        push   ax              ;axが壊れるので退避しておく
        push   bx              ;bxが壊れるので退避しておく
        push   cx              ;cxが壊れるので退避しておく
;
        mov cx,8               ;カウンタを8に設定
        mov dh,al              ;データをdhへ転送
dspb:   xor dl,dl              ;dlをクリア
        rol dx,1               ;dxを左方向へローテート.1桁分がdlへ入る
        add dl,'0'             ;数字に変換するため、dlの数値に文字0を加算
        disp                   ;2進1桁を表示
        loop dspb              ;(cx)←(cx)-1 cxが0までdspからを繰り返す
;
        pop    cx              ;退避させておいたcxを復帰
        pop    bx              ;退避させておいたbxを復帰
        pop    ax              ;退避させておいたaxを復帰
        ret
binB    endp
;-------------------------------------------------
.data
n10b    db  10                     ;数値10byte
n10w    dw  10                     ;数値10word
bcd     db  5 dup(?)               ;10進数を桁数だけ用意
;------------------------------
        end
```

演習問題６

1. 次の制御分岐命令をデータセグメントとコードセグメントを設定して記述せよ.
   (1) byte変数x＝yならばラベルlbaへジャンプする.
   (2) byte変数x＜yならばラベルlbbへ，そうでなければlbaへジャンプする．無符号データとする.
   (3) byte変数x＞＝yならばラベルlbgへ，そうでなければlbeへジャンプする．有符号データとする.
   (4) word変数xとyに有符号データがある．x－yの計算を行い，結果が正か0ならbyte変数sgnを0に，負ならsgnを1に設定する.
   (5) word変数xの値を，変数yの値だけ左へシフトする．(loop命令を使用する)
   (6) byte配列xの10HバイトをFFに設定する．(loop命令を使用する)
   (7) byte配列xの10Hバイトを0に設定する．(loop命令を使用する)
   (8) word変数xとyの乗算を，乗算命令を用いずに実現し，結果を変数aへ保存する．x>y>0とする．ヒント　加算を繰り返す.
   (9) word変数xとyの除算を，除算命令を用いずに実現し，結果を変数aへ保存する．x>y>0とする．ヒント　減算を繰り返す.

2. 次のサブルーチンを示しなさい.
   (1) レジスタALの内容を，3桁の8進数で表示するサブルーチンoctB.
   (2) レジスタAXの内容を，6桁の8進数で表示するサブルーチンoctW.

3. 次のプログラムを記述し，アセンブルして実行せよ.
   (1) keyinと表示してから，キーボードから英字1文字をエコーバックなしで入力し，表示する.
   (2) keyinと表示してから，キーボードから漢字1文字を入力し，表示する.
   (3) キーボードから4文字入力しpaswrdから始まる領域へ保存してから，表示する．入力に際しては文字は表示されないものとする.
   (4) 1～20までの奇数を加算して表示する.
   (5) 1～20までの偶数を加算して表示する.
   (6) byte配列bからの10HバイトをFFに設定し表示する.
   (7) byte配列bからの10Hバイトを00に設定し表示する.
   (8) BCDコードをdec番地から10桁設定し，これを10進数字で画面表示する.

第7章

文字列処理

コンピュータは単に計算をするだけでなく，文字処理を行うこともできます．特に文字列(ストリング：string)の処理は連続的なメモリ操作や繰り返しが多いため，8086ではこのような処理のために専用のストリング命令を用意しています．先に示したように，通常の移動や比較命令などでは，メモリ同士の処理は許されていませんでしたが，これらストリング命令ではメモリ同士の移動や比較が可能となっています．さらに，命令実行後にはインデックスレジスタは自動的にインクリメント，またはデクリメントされるため，連続的なメモリアクセスが可能となっています．

ストリング命令を**表7．1**に示します．文字列の移動，比較とストリングに付けて繰り返しストリング命令を実行する命令プリフィックスがあります．ここでは，8086に用意されているストリング命令について説明します．

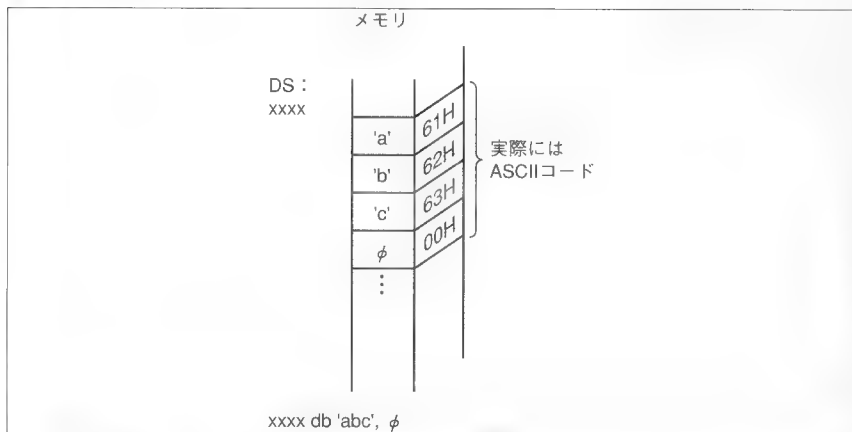
表7．1●ストリング命令

	命令	機能	
文字列移動	MOVS	move byte or word string	文字列移動
	MOVSB	move byte string	文字列をbyte単位で移動
	MOVSW	move word string	文字列をword単位で移動
	LODS	load byte or word string	文字列をロード
	STOS	store byte or word string	文字列をストア
比較	CMPS	compare byte or word string	メモリ上の文字列の比較
	SCAS	scan byte or word string	ALかAXとメモリ上文字列の比較
繰り返し（プリフィックス）	REP	repeat	繰り返し
	REPE/REPZ	repeat while equal/zero	比較結果が＝なら繰り返す
	REPNE/REPNZ	repeat while not equal/not zero	比較結果が≠なら繰り返す

7.1 ストリングの考え方

ここでいう文字列(ストリング)とは，**図7．1**に示すようにメモリ上に連続しておかれた文字の集合のことをいいます．したがって，ストリング命令は連続的なメモリアクセスを効率的に行うための命令として実装されています．これらの命令は，文字列だけでなく任意のバイト列に関して使用することができます．一連の文字列を処理するストリング命令には次のような決まりがあります．

- **文字列からの取り出しは，DS:SI (source index) から行われます．**
- **文字列への書き込みは，ES:DI (destination index) に行います．**
- **文字列を順方向に操作する場合にはcld命令を先に実行し方向フラグDFを0にしておきます．一方，逆方向に操作する場合にはstd命令を先に実行し，方向フラグDFを1にしておきます．**
- **repなどのプレフィックスを付けることにより，同一の処理を繰り返すことができます．**

図7．1●ストリング

7.2 ストリング命令

ストリング命令にはオペランドはなく，暗黙のうちにそのソースとディスティネーションはSIレジスタとDIレジスタが間接的に示すことになっています．必ずソースはSIが，ディスティネーションはDIが指すメモリということになります．これを表7．2に示します．またSIが指すデータはデータセグメントDSにあり，DIが指すデータはエクストラセグメントES内にあります．DSとESに同じアドレスを設定して，データセグメントと同じメモリ領域にすることもよく行われます．

表7．2●ストリング命令でのレジスタ

レジスタ	機能	セグメント
SI	ソース(送り側)　アドレス(offset)	DS
DI	ディスティネーション(受け側)　アドレス（offset）	ES

ここでは各ストリング命令について説明します．以下のストリング命令において，命令の最後の文字が'b'の場合はバイトアクセス，'w'の場合ワードアクセスになります．

・**movsb, movsw (move string)**

[DS:SI]のメモリの内容を[ES:DI]にコピーします(図7．2(a)参照)．

・**cmpsb, cmpsw (compare strings)**

メモリどうしの比較のため，[ES:DI] －[DS:SI]の演算を行い，フラグのみを変化させます．(図7．2(b)参照)

・**scasb，scasb (scan string)**

メモリとアキュムレータの比較のため，AL－[ES:DI]またはAX－[ES:DI]の演算を行い，フラグのみを変化させます(図7．2(c)参照)．

・**lodsb，lodsw (load string)**

[DS:SI] のメモリの内容をALまたはAXに読み出します(図7．2(d)参照)．

・**stosb，stosw (store string)**

ALまたはAXの内容を[ES:DI]のメモリに書き出します(図7．2(e)参照)．

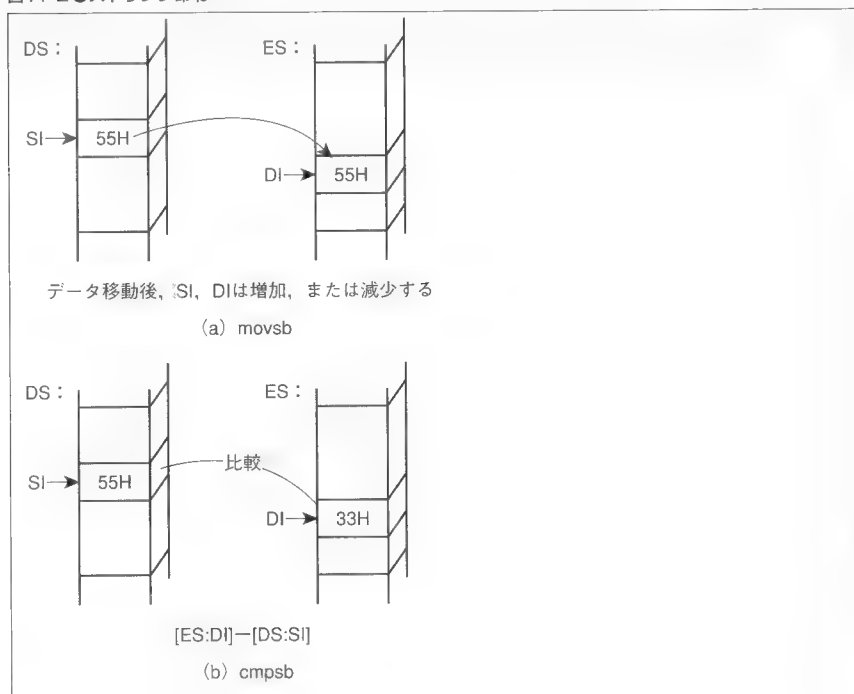
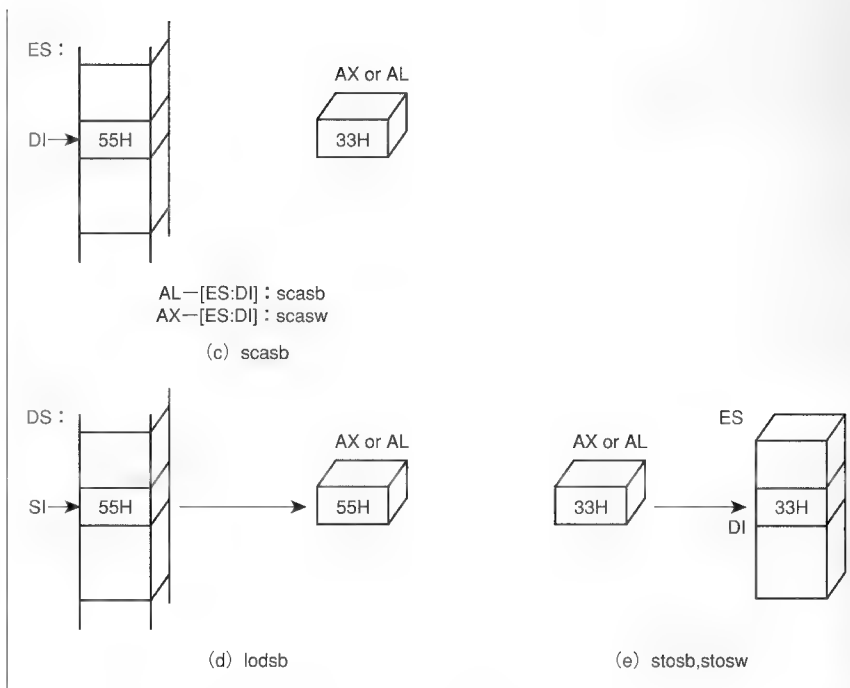
図7．2●ストリング命令

（a）movsb

（b）cmpsb

(c) scasb

(d) lodsb

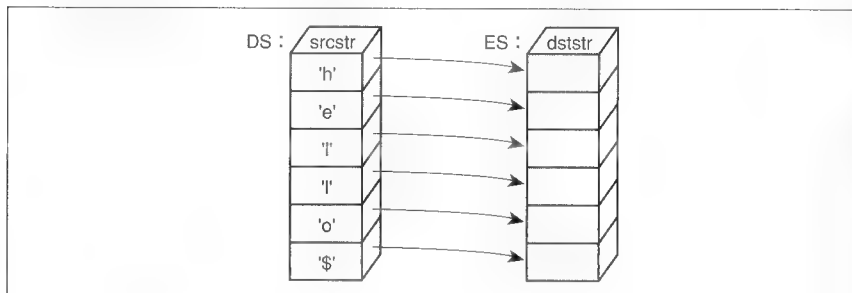
(e) stosb,stosw

また，これらの命令は繰り返して利用されることが多いため，rep，repe，repneなどの命令プリフィックスを頭に付けることができます．この命令プリフィックスを付けた場合には，命令を実行するたびにCXを減じていき，CXが0になったときに終了します．そのため，命令を実行する前処理として，カウンタとして使用するCXレジスタに繰り返す回数をセットしておきます．

7.3 ストリング命令の使用例

■(a)文字列のコピー

図7．3に示すように転送元(srcstr)にある文字列を転送先のメモリにコピーするプログラムを**リスト7．1**に示します．ここでは文字数がすでにわかっている場合の例です．行5:～9:まではセグメントの設定をしています．文字列処理命令はESも使用することに注意して下さい．ここでは同一データセグメント内

のコピーなので，同じセグメントアドレスを代入しています．行11:および行12:は転送元と転送先のアドレスをそれぞれ，SIとDIに代入しています．行13:では'hello$'の文字数である6を代入しています．この例では文字数がわかっているので，行15:ではrepによってmovsbを指定回数だけ繰り返しています．最後にコピーができていることを確認するために，dispsマクロで転送先のデータを確認しています．

図7．3●文字列のコピー

リスト7．1●文字列のコピー

```
 1: .model small
 2: ;
 3:         include             iomac.inc
 4: ;-----------------------------------
 5: .code ;コードセグメント
 6: start:
 7:         mov ax,@data        ;@dataはセグメントアドレス
 8:         mov ds,ax           ;DSへデータセグメントの設定
 9:         mov es,ax           ;ESもDSと同じ値に設定
10: ;-----------------------
11:         lea si,srcstr       ;転送元アドレスを設定
12:         lea di,dststr       ;転送先アドレスを設定
13:         mov cx,6            ;文字数を設定
14:         cld                 ;文字列の操作を順方向に
15:         rep movsb           ;指定された文字数分だけコピー
16: ;
17:         lea dx,dststr
18:         disps               ;文字列表示マクロ
19:         exit                ;プロセス終了のマクロ
20: ;---------------------------------------
21: .data                       ;データセグメント
```

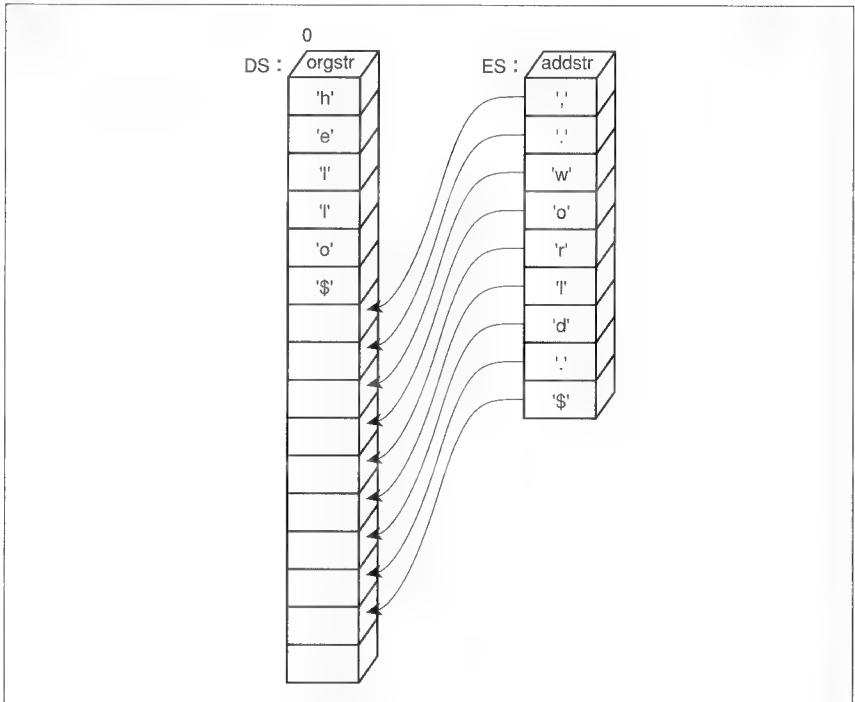
```
22: srcstr db 'hello$'          ;転送元データ
23: dststr db 6 dup(?)          ;転送先
24: ;---------------------------------------------------
25: .stack 100H                 ;スタックセグメント
26: ;---------------------------------------------------
27:         end start
```

使用例

```
C:¥prg>movs
hello
```

■(b)文字列の連結

図7．4に示すようにある文字列の後ろに別の文字列をコピーし，連結した文字列を作成するプログラムを**リスト7．2**に示します．このためにはまずオリジナルの文字列の最後を検索する必要があります．この処理は行15:のrepne scasbにて行われます．行11:～14:まではそのための準備をしています．(a)の場合と異なり文字数はチェックしないので，CXには取りうる最大値を設定してあります．また，ALには文字列の終端文字'$'を設定しています．検索終了後，DIは終端文字の次のアドレスを示しているので，行16:ではDIの値を1つ戻して，その部分から文字列を追加します．行17:以降は(a)とほぼ同じ処理ですが，文字数はわかっていないので，終端文字かどうかの比較を行19:にて行っています．

図7．4●文字列の追加

リスト7．2●文字列の連結

```
 1: .model small
 2: ;-----------------------------------
 3:         include  iomac.inc
 4: ;-----------------------------------
 5: .code                       ;コードセグメント
 6: start:
 7:         mov ax,@data        ; @dataはセグメントアドレス
 8:         mov ds,ax           ; DSへデータセグメントの設定
 9:         mov es,ax           ; ESもDSと同じ値に設定
10: ;-------------------------
11:         lea di,orgstr       ;オリジナルの文字列を設定
12:         mov al,'$'          ;比較のため終端文字を al に代入
13:         mov cx,65535        ;カウンタは最大値(65535)
14:         cld                 ;文字列の操作を順方向に
15:         repne scasb         ;終端文字と等しくなるまで繰り返し
16:         dec di              ;1 文字戻す
17:         lea si,addstr       ;追加する文字列を設定
18: loop1:  movsb               ;1 文字ずつコピー
```

```
19:         cmp al,[si-1]       ;1 文字前が終端文字か?
20:         jne loop1           ;そうでなければ繰り返し
21:         lea dx,orgstr       ;オリジナルの文字列を設定
22:         disps               ;文字列表示マクロ
23:         exit                ;プロセス終了のマクロ
24: ;--------------------------------------
25: .data
26: orgstr db 'hello$'          ;オリジナル文字列
27:         db 10 dup (?)       ;追加用メモリ空間
28: addstr db ', world.$'       ;追加文字列
29: ;------------------------------------------------
30: .stack 100H                     ;スタックセグメント
31: ;------------------------------------------------
32:         end start
```

実行例

```
C:¥prc>scas
hello, world.
```


■(c)ASCII－数字変換

ASCIIコードを数字に変換するプログラムを**リスト7．3**に示します．キーボードから入力された文字が，'0'から'9'または'a'から'f'だった場合に対応する数字をaxに代入し，そうでなかった場合には－1を代入します．行12:でキーボードから1文字入力された文字のアスキーコードがalに代入されます．このアスキーコードから数字に変換するために，データセグメントにある16進テーブルの検索を行います．

リスト7．3●ASCIIコード－数字変換

```
 1: .model small
 2: ;
 3: extrn hexW:near
 4: include  iomac.inc
 5: ;--------------------------------------
 6: .code                       ;コードセグメント
 7: start:
 8:         mov ax,@data        ;@dataはセグメントアドレス
 9:         mov ds,ax           ;DSへデータセグメントの設定
```

```
10:         mov es,ax           ;ESもDSと同じ値に設定
11: ;---------------------------
12:         keyin               ;1 文字入力
13:         lea di,table        ;16 進テーブル
14:         mov cx,11h          ;16 (16 進数) + 1 (その他)
15:         cld                 ;文字列の操作を順方向に
16:         repne scasb         ;16 進テーブルを検索
17:         dec cx              ;cx = 0 でないときはスキップ
18:         crlf                ;改行する (マクロ)
19:         mov ax,cx           ;表示のため
20:         call hexW           ;16進表示
21:         exit                ;プロセス終了
22: ;-------------------------------------------------
23: .data                       ;データセグメント
24: table   db 'fedcba9876543210O'
25: ;-------------------------------------------------
26: .stack 100H                 ;スタックセグメント
27: ;-------------------------------------------------
28:         end start
```

実行例

```
C:¥prc>ascnm
3
0003
C:¥prc>ascnm
f
000F
C:¥prc>ascnm
F
FFFF
```

演習問題7

1. 次の文字列処理命令をコードセグメントとデータ・エクストラセグメントを設定して記述せよ.

   (1) byte配列xの文字列(10文字)をyへ移動する.

   (2) byte配列xの文字列(10文字)をyへ逆順で移動する.

   (3) tbl番地から英大文字(A～Z)が26文字格納されている. alpa番地においた適当な英小文字10文字を, テーブルを用いて英大文字に変換する.

   (4) 変数xにあらかじめ置かれた文字1文字が, 配列alphの文字列(10文字)中にあるかを検索する. 文字列中にあるならその要素番号を, ないなら0～9でない数をnに設定する.

2. 次のプログラムを記述し, アセンブルして実行せよ.

   (1) 配列alphにアルファベット26文字をおいて, これを表示せよ.

   (2) 配列alphにアルファベット26文字をおいて, これを逆から(zyx・・・)表示せよ.

   (3) 配列alphに適当な文字列(10文字)をおいて, キーボード入力した1文字がalph中にあるかを検索する. alph中にあるならその要素番号を, ないなら10以上の数を表示する.

第 8 章

入出力命令と割り込み命令

コンピュータはデータを入出力するために，入出力装置とのインタフェースを持ちます．8086ではデータ入出力のための入出力命令を持っています．またこれら入出力装置の動きを制御するために，ハード的な割り込み機構を持っています．現在のコンピュータではこの割り込みは重要な働きをしています．8086には，ハード的に割り込みが起こった場合と同じ働きをする割り込み命令(ソフトウェア割り込み)を持っています．

ここでは，プロセッサが入出力装置とデータをやり取りをするために使用する入出力命令と割り込み命令について説明します．

8.1 入出力命令

プロセッサに外部機器を接続するには2つの方法があります．1つは，専用の入出力ポート(以下I/Oポートと呼びます)を使う場合であり，もう1つは主記憶(メモリ)の一部をI/Oポートとして取り扱う場合です．

8086では**図8．1(a)**に示すように主記憶の他に64kバイト分のI/Oポートが存在します．入出力命令を**表8．1**に示します．

表8．1●入出力命令

分類	命令	機	能
入出力	IN	input byte or word	データ入力
	OUT	output byte or word	データ出力

入出力命令の使用例を**リスト8．1**に示します．周辺機器を0aH番地のI/Oポートに接続した場合には，

- **in al,0aH　または　in ax,0aH　　I/Oポートを定数で指定**
  in al,dx　　または　in ax,dx　　　I/OポートをDXで指定
- **out 0aH,al　または　out 0aH,ax　　I/Oポートを定数で指定**
  out dx,al　　または　out dx,ax　　　I/OポートをDXで指定

という入出力命令を使用します．I/Oポートとやり取りできるのはアキュムレータのみであり，AXで指定した場合は16ビット，ALで指定した場合は8ビットの転送が行われます．

ポート番号は即値0aHで指定するかわりに，DXレジスタで指定することもできます．定数で指定する場合は0～255番地まで，DXで指定する場合は0～64k番地まで指定することができます．

リスト8．1●in，out命令の使用例

```
1: .code
2: in al,0aH              ;ポート0aHから，ALへ入力(byte)
3: in ax,0aH              ;ポート0aHから，AXへ入力(word)
4: mov dx,0aH             ;DXに0aHを設定
```

```
 5: in al,dx            ;DXで示すポートから，ALへ入力(byte)
 6: in ax,dx            ;DXで示すポートから，AXへ入力(word)
 7:                     ;
 8: out 0aH,al          ;ALから，ポート0aHへ出力(byte)
 9: out 0aH,ax          ;AXから，ポート0aHへ出力(word)
10: mov dx,0aH          ;DXに0aHを設定
11: out dx,al           ;ALから，DXで示すポートへ出力(byte)
12: out dx,ax           ;AXから，DXで示すポートへ出力(word)
```

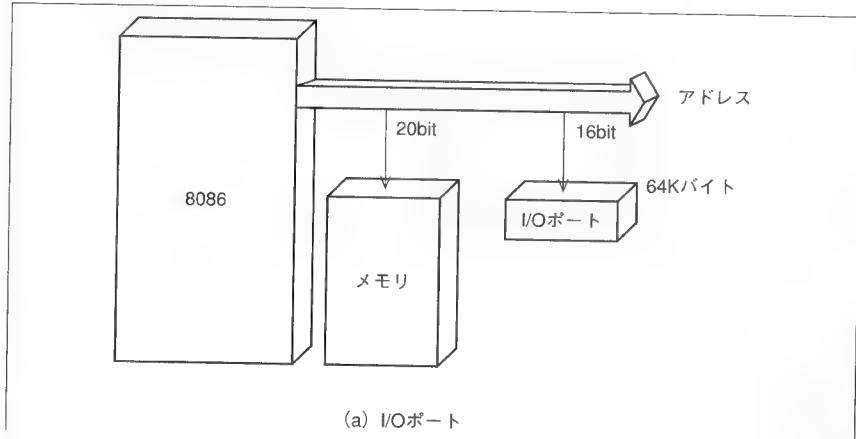
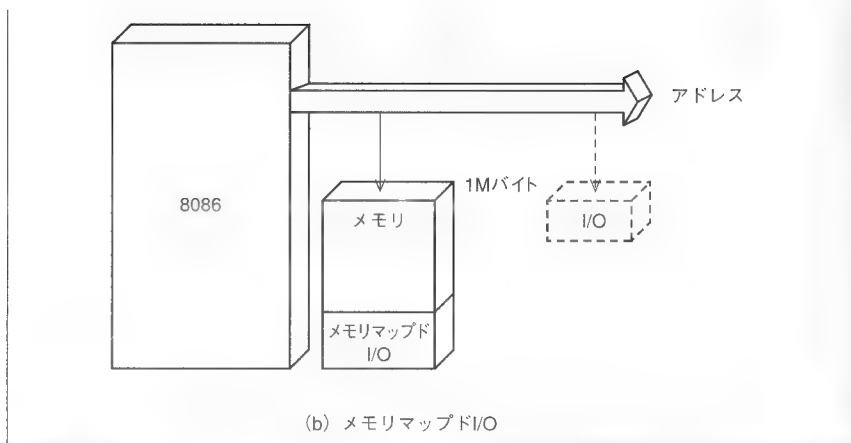
表8．2●入出力命令

分類	命令	機能	
割り込み	INT	interrupt	割り込み
	INTO	interrupt if overflow	オーバフロー割り込み
	IRET	interrupt return	割り込み処理からの復帰

一方，図8．1(b)に示すように主記憶の一部に周辺機器を割り当て，I/Oポートのように利用する方法があります．これをメモリマップドI/O(Memory mapped I/O)といいます．この方法では，主記憶の一部をI/Oポートとするため，記憶領域として使用できるメモリは少なくなります．しかしながら，主記憶をアクセスするアドレッシングが利用できるため，in命令やout命令のようなレジスタの制限が少なくなります．

図8．1●I/Oポートの構成方法

(a) I/Oポート

(b) メモリマップドI/O

8.2 8086の割り込み

通常の処理を行っているときに，それを一時中断し，他の処理を優先的に実行することを割り込み処理といいます．例として，キーボードからの入力について考えてみます．もし，CPUが直接キーボードを監視したりすると，周期的にキーボードが接続されている入出力ポートを調べることになります．この周期が短い場合，本来の目的である計算の効率が低くなります．逆に効率をよくするために周期を長くすると，キーの入力を取りこぼす可能性があります．このような場合に有効なのがここで説明する割り込みです．

割り込みを使用する場合，CPUとは別に周辺機器のコントロール用プロセッサを載せます．このプロセッサは周辺機器からの応答をCPUに伝えるべきであると判断した場合，CPUに割り込みをかけます．割り込みがかかると，CPUはフラグレジスタおよび割り込み終了後に実行すべき命令のアドレス(現在のPC値)をスタックに積み，割り込み処理ルーチンに飛びます．割り込み処理が終了した後，保存してあるフラグレジスタを復帰させ，実行していた元のプログラムに戻ります．この結果，CPUは周辺機器のことを気にすることなく，本来の計算を行うことができます．

割り込み処理中にさらに割り込みがかかることは望ましくありません．8086には割り込みレベルという概念がないため，割り込みを許可するか許可しない

かのみ選択できます．この制御は，**図8．2**に示す状態フラグ中のIフラグで行います．フラグ操作に関する命令を**表8．3**に示します．ここでstiとcli命令は次のような働きをします．

sti　I=1にセットします(SeT Interrupt).
割り込み許可モードになります.
cli　I=0 にクリアします(CLear Interrupt).
割り込み禁止モードになります.

通常は割り込み処理ルーチンに入った時点で，cliを実行します．この割り込み禁止モードでは，NMI(non-maskable interrupt)と呼ばれる禁止できない割り込み以外の割り込みは起こらなくなります．

表8．3●割り込みフラグ

分類	命令	機能	
フラグ操作	STI	set interrupt enable flag	割り込み許可フラグをセット
	CLI	clear interrupt enable flag	割り込み許可フラグをクリア

図8．2●状態フラグ

コンピュータには多くの周辺機器が接続されますので，割り込みの種類は1つだけではありません．したがって，割り込みがかかった場合には，その機器に対応した割り込み番号(0～255)が同時に知らされます．この割り込み番号と割り込みルーチンを結び付けるのが，割り込みベクトルです．これは，セグメント0の0番地に置かれたジャンプテーブルで，割り込み番号の順に割り込み処理ルーチンの先頭アドレスを記述したものです．たとえば，NMIであれば割り込み番号が2なので，8～11番地に書かれたアドレスを割り込み処理ルーチンと

して呼び出します．

ハードウェアによる割り込みとしては，他にも0による割り算，デバッガのためのシングルステップ実行，オーバフロー割り込みなどがあります．

8.3 割り込み命令

上記のハードウェア割り込みの他に，ソフトウェアからも割り込みを発生することができます．**表8．2**に割り込み命令を示してあります．割り込み命令としては，

- **int 3**　　**3番の割り込みを起こす．**
- **int n**　　**n番の割り込みを起こす．**
- **into**　　**オーバフローが起きていたら割り込みを起こす．**

の3種類が存在します．int nはn番目の割り込みを起こす命令です．割り込み番号nと機能の割り振りはそのハードウェア，OSによって異なります．また，int 3だけは特別に1バイトの命令で，プログラムに埋め込みやすいために，デバッガと呼ばれるプログラムで使われます．intoは演算命令実行後に行うもので，演算においてオーバフローが起こっていた場合にint 4と同じ割り込みが起こります．

多くのOSでは，このint命令をカーネルのサブルーチン呼び出しに利用しています．MS-DOSには割り込み命令を用いてシステムコールを行うことができます．このMS-DOSシステムコールの機能については，10章で説明します．

演習問題 8

1. メモリマップド I/O を用いる利点・欠点を述べよ.

2. int 3 の割り込みが起きたとき実行される割り込み処理ルーチンの先頭アドレスは，メモリ空間のどこに書かれているか述べよ.

3. 割り込み処理内で，割り込みを禁止する理由を述べよ.

第 9 章

疑似命令とマクロ命令

疑似命令(directive)は実際に機械語を作り出すのではなく，アセンブリ言語の処理系に指令を与えたり，情報を伝えるものです．疑似命令によって実際の機械語が生成されることはありませんが，変数や配列の領域がとられて，その初期値を与えることはできます．これまでにいくつかの疑似命令を取り扱ってきましたが，本章ではこれらをまとめて説明することにします．

マクロ命令は一連の命令をひとまとめにして，これを繰り返し用いようとするものです．プロシージャと同じように思われますが，プロシージャと異なり，利用されるプログラム部分に実際の機械語が埋め込まれます．したがって，プログラムに記述された数だけ機械語が生成されていることになります．

本章では，疑似命令とマクロ命令について概要を説明します．

9.1 疑似命令

疑似命令はアセンブラに指令を出したり情報を伝えるものですが，MASMではいくつかに分類することができます．**表9．1**に本書で用いる疑似命令を示します．実際にMASMの疑似命令は非常に多く，これらは巻末の付録に示しました．

表9．1●本書で用いる疑似命令

種　別	疑似命令	機　能	使用例
プロセッサ指定	.8086 .386	8086命令をアセンブル可能にする．上位プロセッサ命令は禁止． 80386非特権命令をアセンブル可能にする	.8086 .386
簡略化セグメント	.model .code .data .stack .const .data? assume	メモリモデルを定義．Small,compact,medinum,large,hugeなど コードセグメントの始まり 初期化データセグメントの始まり スタックセグメントの始まり 定数データセグメントの始まり 未初期化データセグメントの始まり セグメントとレジスタの対応付け	.model small .code .data .stack assume cs:_text,ds:dgroup
データ領域割当	db dw dd dq	byteデータ領域の割り当て word(2byte)データ領域の割り当て double word(4byte)データ領域の割り当て quad word(8byte)データ領域の割り当て	db 'a' dw 'ab' dd 'abcd' dq 'abcdefgh'
コードラベル	proc endp label	プロシージャの始まり プロシージャの終わり ラベルの定義	pro1 proc pro1 endp label lbl1 byte
有効範囲	extrn public	外部変数、ラベルの定義 変数、ラベルを他のプロシージャから参照可能にする	extrn val1 public val1
マクロ	macro endm	マクロブロックの始まり マクロブロックの終わり	mc1 macro par1,par2 endm
等価記号	equ	名称に式を割り当て	decmal equ 10
演算子	dup offset	初期値のコピー オフセット値を返す	x db 10 dup(?) offset x

■プロセッサ指定疑似命令

プロセッサ指定疑似命令は，アセンブラにどのようなコードを生成するかを指令します．

.8086

アセンブラに8086命令でコードを生成するように指示します．80386やPeniumなど上位プロセッサの命令は禁止されます．

.386

アセンブラに80386の非特権命令をアセンブルするように指示します．特権命令を含めたすべての命令をアセンブルする場合は.386Pとします．

■簡略化セグメント疑似命令

簡略化セグメント疑似命令は，複雑であったMASMのセグメント定義をわかりやすくするために作られた疑似命令です．これら疑似命令を用いると実際のプログラムのセグメント関連の記述が簡単になります．

.model　　.MODEL　memorymodel

プロセッサが用いるメモリモデルを定義します．メモリモデルmemorymodelとしてはsmall,compact,medium,large,huge,flatのいずれかを指定できます．スタンドアロンのアセンブラプログラムにはsmallモデルが一般的でしょう．メモリモデルを表9．2に示します．

表9．2●メモリモデル

モデル	コード	データ	レジスタの仮定	機　能
SMALL	near	near	cs=_text ds=ss=DGROUP	コードは単一セグメント内にあり、全データはDGROUPに結合される．スタンドアロンのプログラムに多く用いられる．
MEDIUM	far	near	cs=<module>_text ds=ss=DGROUP	コードは複数セグメントを使用し、全データはDGROUPに結合される．
COMPACT	near	far	cs=_text ds=ss=DGROUP	コードは単一セグメント内にあり、全データはDGROUPに結合される．データ参照にfarポインタが用いられる．
LARGE	far	far	cs=<module>_text ds=ss=DGROUP	コードは複数セグメントを使用し、全データはDGROUPに結合される．データ参照にfarポインタが用いられる．
HUGE	far	far	cs=<module>_text ds=ss=DGROUP	コードは複数セグメントを使用し、全データはDGROUPに結合される．データ参照にfarポインタが用いられる．
FLAT	near	near	cs=_text ds=ss=flat	SMALLモデルと同様だが、32bitフラットメモリモデルが用いられる．

smallモデルは，コードとデータともに64kバイトの各セグメントに配置されます．コードとデータ共にnear(16ビット)アドレスでアクセス可能です．コー

ドは単一セグメント内にあり，セグメント名称として_textと名付けられます．.model small簡略化疑似命令では自動的に次のステートメントが生成されますが，リスト上には表現されません．

```
dgroup group _data,const,_bss,stack
assume cs:_text,ds:dgroup,ss:dgroup
```

最初の宣言でdgroupがグループで，その構成はデータ，定数，スタック領域であると定義されます．次のassume宣言で，コードセグメントcsは_text，データセグメントdsとスタックセグメントssは共にdgroupであると宣言されています．このdgroupのセグメント内で定義された変数やラベルは，定義されているセグメントの先頭アドレスからでなく，グループの先頭からのアドレス相対で参照されることになります．なお名前の関連は非常にわかりにくいところですが，dgroupのアドレスは等価記号@dataで表現します．

.code

.MODEL疑似命令を用いている場合に，コードセグメントの始まりを定義します．セグメントの先頭番地は_textとなります．

.data

.MODEL疑似命令を用いている場合に，データセグメントの始まりを示します．.model疑似命令を用いた場合，dgroupグループ内に配置されます．dgroupのセグメント内の変数はグループの先頭からの相対アドレスで示されることになります．dgroupのアドレスは@dataとなります．

.stack

.MODEL疑似命令を用いている場合に，スタックセグメントの始まりを示します．.model疑似命令を用いた場合，dgroupグループ内に配置されます．stack内のデータは，スタックセグメントの先頭でなく，グループdgroupの先頭からの相対アドレスで示されることになります．dgroupのアドレスは@dataとなります．

.data?

.MODEL疑似命令を用いている場合に，未初期化nearデータセグメントを定義します．

assume　　ASSUME　segreg:name

セグメントとセグメントレジスタの対応付けのための宣言です．nameで指定されたセグメントが，セグメントレジスタsegregをデフォルトとして割り当てます．この疑似命令はアセンブラの処理系に，コード，データなどのセグメントが実際にどのように割り当てられているかを情報として与えるものです．セグメントレジスタへその先頭アドレスを渡しているのではないので注意が必要です．

■データ領域割り当て疑似命令

db　　name　DB　init,[init]

nameと名付けたバイト(byte)単位のデータ領域を定義し，initで初期化します．複数の要素を定義可能で，次のように使用します．

```
var       db       11
```

varと名付けた変数を定義し，その初期値を11とします．

```
string       db       'abcdefg'
```

stringと名付けた文字配列を定義し，' '(または" ")で囲まれた文字列で初期化します．要素数は文字数で決まります．

```
array       db       100      dup(0)
```

arrayと名付けた配列(要素数100)を定義し，全要素を0で初期化します．

```
narray       db       1,2,3,4
```

narrayと名付けた配列(要素数4)を定義し，初期値をそれぞれ1,2,3,4とします．

dw　　name　DB　init,[init]

nameと名付けたワード(word 2byte)単位のデータ領域を定義し，initで初期化します．

dd　　name　DD　init,[init]

nameと名付けたダブルワード(double word 4byte)単位のデータ領域を定義し，initで初期化します．

■コードラベル疑似命令

プロシージャやラベルを定義する疑似命令です．

proc　　name　PROC　[NEAR|FAR]

nameと名付けたプロシージャ(サブルーチン)の始まりを定義します．そのプロシージャがNEARかFARかの指定を行うこともできます．

PROC疑似命令は，次の形式となります．

```
pname   proc   far
```

pnameはプロシージャ名で，farは4バイトアドレスでコールされることを示しています．2バイトアドレスでコールされる場合は，nearで表現します．デフォルトはnearなので，何も付けないとnearとなります．

endp　　name　ENDP

プロシージャの終了を示します．次のようにprocで宣言した名前をもう一度用います．procとendpでは名称が対になっている必要があります．

```
pname proc
  .
  .
pname endp
```

label　　name　LABEL　type

nameと名付けたラベルをtype型として定義します．たとえば

```
lbl1 label byte
```

では，lbl1という名称のラベルを，byteとして用意します．labelはそれが定義されたファイル内からのアクセスが原則ですが，public疑似命令で宣言すれば，他のモジュール(異なったファイル内のプログラム)からもアクセス可能となります．

extrn　　EXTRN　name:type　[,name:type]

他のモジュール内にある外部シンボル(変数，配列，ラベルなどに付けられた名称)nameの参照を宣言します．外部シンボルは他のモジュール内で，public宣言されている必要があります．

public　　PUBLIC　name　[,name]

モジュール内のシンボルnameを，他のモジュールから参照可能にします．定義したシンボルを，他のモジュールから参照するためには，参照するモジュールでそのシンボルをextrn宣言する必要があります．

macro　　name　MACRO　[param1[,param2].....]

nameで引用されるマクロブロックを定義します．param1....はマクロの引用時に与えられた引数として置換されます．詳細は次節を参照してください．

endm **ENDM**

マクロ定義の最後を示します．これについても詳細は次節を参照してください．

equ **name EQU expression**

シンボルnameに式expressionの値を与えます．この宣言後，nameに出会うとアセンブラ処理系は，nameをexpressionの値で置き換えます．たとえば

```
decemal    equ    10
```

では，decemalが現れるたびに，それが10に置き換えられ処理されます．

offset **OFFSET exprssion**

式expressionのオフセット値を返します．たとえば

```
offset    x
```

では，xが変数として定義されていれば，そのオフセット値を返します．

dup **count DUP (initvalue)**

count個の初期値の宣言をします．たとえば

```
x    db    100    dup(?)
```

では，byte配列xを要素数100個として領域をとり，これを初期化しないことを意味します．

9.2 マクロ命令

ある目的のプログラムを記述する際，同じ働きのプログラムが再三必要になる場合がよくあります．このような場合，アセンブリ言語のプログラムでは，記

述を簡単化する方法が2つあります．1つは，同じ働きをするプログラムを，サブルーチンとして必要になるたびに呼び出す方法です．

他の1つは，マクロ命令を用いる方法です．マクロ命令は一連の命令に名前を付けたものですから，サブルーチンとよく似た形となります．サブルーチンのようにcall命令で呼ばれるのではなく，一連の命令が使用される部分に埋め込まれます．したがって10回マクロ命令が引用されれば，ループ内で引用されるのではない限り，10個の同じ命令群がプログラム内に存在することになります．

サブルーチンでは何度callされようと，実際に存在する命令群は1つだけですから，メモリ効率はサブルーチンの方がよいといえるでしょう．しかし，call命令が実行されるときには，リターン番地がスタックへ保存されるなど，その手続きに時間がかかります．

マクロ命令を使用する場合は，引用されたその部分にコードが埋め込まれますから，呼び出し手続きに時間を費やすことはありません．メモリ効率には目をつむってでも，実行速度を高めたいときには，サブルーチンを用いるよりマクロ定義を用いることになります．

マクロ命令は一連の命令に名前を付け，その名前を引用することでその命令群を展開可能にします．

■マクロ命令の定義

マクロ命令の定義方法を下に示します．mov,add,addの3命令をマクロ名addupとして定義します．macroとendmは疑似命令で，この間に必要となる命令を書きます．ad1,ad2,ad3は仮パラメータで，addupを引用するときに付けられた実パラメータにより，置き換えられます．

```
addup   macro   ad1,ad2,ad3
        mov     ax,ad1
        add     ax,ad2
        add     ax,ad3
        endm
```

■マクロの使用

マクロ命令の使用法を下に示します．addupがマクロ命令で，bx,2,countが実パラメータです．

```
        addup   bx,2,count
```

■マクロ展開

上記のようにaddupが引用されると，3命令mov,add,addが展開されます．仮パラメータad1,ad2,ad3が，それぞれ実パラメータbx,2,countに置き換えられます．

```
        mov     ax,bx
        add     ax,2
        add     ax,count
```

9.3 完全なセグメント定義によるプログラム

ここでは，完全なセグメント定義をしたプログラムを紹介します．**リスト9．1**にスタック命令のプログラム例を示します．このプログラムは**リスト3．11**のプログラムを完全なプログラム定義に変更したものです．

各セグメントはsegment疑似命令によって定義します．これらセグメントがどこに所属するかを，行1:のassume疑似命令でアセンブラ処理系に情報として与えます．行3:～5:でmydataと名付けたデータセグメントを，行7:～10:でmystackと名付けたスタック領域を定義しています．bottom label wordでスタックの底を示すラベルを定義していますが，このラベルはスタックポインタSPの初期値を設定するのに用います．

行14:～18:はセグメントレジスタの設定とSPの初期設定を行います．データセグメントレジスタDSにmydataを，スタックセグメントレジスタにmystack

を与えています．またlea sp,bottomでSPを初期化しています．完全なセグメント定義を行う場合は，コードセグメントレジスタCSを除くセグメントレジスタの設定とスタックポインタSPの設定が必要となります．

このプログラムの実行例は，**リスト3．11**に示したプログラムと同じ結果が得られています．

リスト9．1●完全なセグメント定義　スタック

```
 1:        assume cs:mycode,ds:mydata,ss:mystack;処理系へのセグメント情報
 2: ;-------------------------------------------------------------
 3: mydata  segment             ;データセグメント
 4: w  dw  'ab'                 ;変数w
 5: mydata  ends                ;データセグメントはここまで
 6: ;------------------------------------------
 7: mystack segment stack       ;スタックセグメント
 8:    dw  100H dup (?)         ;100ワードの領域をとる
 9: bottom  label word          ;スタックボトム
10: mystack ends                ;スタックセグメントの終了
11: ;------------------------
12: mycode  segment             ;コードセグメント
13: stackprg    proc far        ;メインプログラム(far)
14:         mov ax,mydata       ;データセグメントのアドレス
15:         mov ds,ax           ;DSを設定
16:         mov ax,mystack      ;スタックセグメントのアドレス
17:         mov ss,ax           ;SSを設定
18:         lea sp,bottom       ;スタックポインタSPを設定
19: ;-----------------------------
20:         push w              ;(sp)←(sp)-2,((sp)+1:(sp))←(w)
21:         pop dx              ;(ax)←((sp)+1:(sp))←(ax),(sp)←(sp)+2
22: ;-----------------------------
23:         xchg dl,dh          ;DLとDHの内容を交換
24:         mov ah,2            ;ファンクション2(1文字表示)
25:         int 21h             ;MS-DOSフファンクションをコール
26: ;
27:         xchg dl,dh          ;DLとDHの内容を交換
28:         mov ah,2            ;ファンクション2(1文字表示)
29:         int 21h             ;MS-DOSファンクションをコール
30: ;
31:         mov ax,4c00h        ;ファンクション4C.リターンコード0
32:         int 21h             ;MS-DOSファンクションコール(復帰)
33: stackprg    endp
34: mycode  ends
```

```
35: ;
36:         end stackprg


実行例
    C:¥prg>stack
    ab
```

リスト9．2に完全なセグメント定義をした文字列表示プログラム例を示します．このプログラムは11章の**リスト11．5**のプログラムを完全なセグメント定義で記述したものです．

各セグメントに名前を付けてその所属をassume疑似命令で定義し，アセンブラ処理系に情報として与えているのは，前例と同様です．

このプログラムの実行例でも，**リスト11．5**に示したプログラムと同じ結果が得られています．

リスト9．2●完全なセグメント定義　文字列表示

```
 1: assume cs:mycode,ds:mydata,ss:mystack;処理系へのセグメント情報
 2: ;
 3: mydata  segment              ;データセグメント
 4:  strg db 'Hello ,x86アセンブラ!',0dh,0ah,'$';文字列、CRとLF ;$はデータ終了印
 5: mydata  ends                 ;データセグメントの終了
 6: ;-----------------------------------------
 7: mystack segment stack        ;スタックセグメント
 8:         dw 100 dup(?)        ;100ワードの領域をとる
 9:  bottom label word           ;スタックボトム
10: mystack ends                 ;スタックセグメントの終了
11: ;-----------------------------------------
12: mycode  segment              ;コードセグメント
13: kdisp   proc far             ;メインプログラム(far)
14:         mov ax,mydata        ;データセグメントのアドレス
15:         mov ds,ax            ;CSを設定
16:         mov ax,mystack       ;スタックセグメントのアドレス
17:         mov ss,ax            ;SSを設定
18:         lea sp,bottom        ;スタックポインタSPを設定
19: ;--------------------------------
20:         lea dx,strg          ;文字列番地ををdxレジスタへ
21:         call disps           ;文字列表示プログラムの呼び出し
```

```
22: ;
23:         mov ax,4c00h        ;ファンクション番号4C.リターンコード0
24:         int 21h             ;MS-DOSファンクションコール(復帰)
25: kdisp   endp
26: ;--------------------------------
27: disps   proc near           ;文字列を表示するサブルーチン
28:         mov ah,9h           ;ファンクション番号9
29:         int 21h             ;MS-DOSファンクションコール(割込み)
30:         ret                 ;復帰
31: disps   endp
32: ;--------------------------------
33: mycode  ends                ;コードセグメントの終了
34: ;
35:         end kdisp           ;kdispからプログラムを開始する
```

実行例

```
C:¥prg>kdisp
Hello ,x86アセンブラ!
```

演習問題 9

1. 次のプログラムを記述せよ.

(1) MS-DOSシステムコール"日付の読み出し"（2AH）をマクロ化する.

(2) MS-DOSシステムコール"時刻の読み出し"（2CH）をマクロ化する.

(3) 3章の**リスト3. 2**の移動命令プログラムを完全なセグメント定義に変更する.

(4) 4章の**リスト4. 3**の加減算命令のプログラムを完全なセグメント定義に変更する.

(5) 5章の**リスト5. 2**のxor命令のプログラムを完全なセグメント定義に変更する.

第10章

MS-DOSシステムコール

この章までのプログラム例では，データの入力や表示をするために，詳細な説明なしでMS-DOSシステムコールを用いてきました．この章ではMS-DOSシステムコールについて説明します．

アセンブリ言語のプログラムを実行する場合，ターゲットとなるコンピュータでは，何らかの入出力装置があるでしょう．コンピュータでは，inやout命令を用いて直接入出力装置からデータ入出力をするためには，そのIO番地や構成を知る必要があります．しかし，それぞれのコンピュータで，ハードウェア構成が異なり，IO番地も異なっています．オペレーティングシステムの持つ基本的なサブルーチンを用いることができれば，マシン構成を知ることなくこれらの入出力が可能になります．

10.1 システムコールの方法

本書でのプログラムは，基本的に基礎的なOSであるMS-DOS上で動作するよう記述してきました．ユーザプログラム実行の際には，MS-DOSシステムコールをすることで，MS-DOSのサブルーチン群を用いることができます．

MS-DOSが準備しているシステムコールは，バージョンによって異なりますが，キー入力やディスプレイ表示など，約70種のほどの基本サブルーチンを呼び出すことができます．本書で利用するMS-DOSファンクションを**表10．1**に示します．

表10．1●MS-DOSファンクション(一部)

番号	機能	呼び出し	戻り値
01	キーボード入力（エコー付き）		AL　入力文字
02	1文字出力	DL←出力文字	
07	コンソール直接入力		AL　入力文字
08	キーボード入力（エコーなし）		AL　入力文字
09	文字列出力	DX←文字列の先頭アドレス	
0A	バッファへのキーボード入力	DX←バッファの先頭アドレス	
4C	プロセスの終了	AL←リターンコード	

システムコールをするには，割り込み命令を用います．AHレジスタにファンクション番号を設定し，出力の場合はDLまたはDXレジスタにデータまたはアドレスを設定します．その後，int 21Hの割り込み命令を用いてシステムのサブルーチンを呼び出します．

■システムコールの方法

AH	ファンクション番号を設定
AL	入力データ(入力時)

DLまたはDX	出力データまたはアドレス
int 21H	ソフトウェア割り込みでMS-DOSシステムCALL

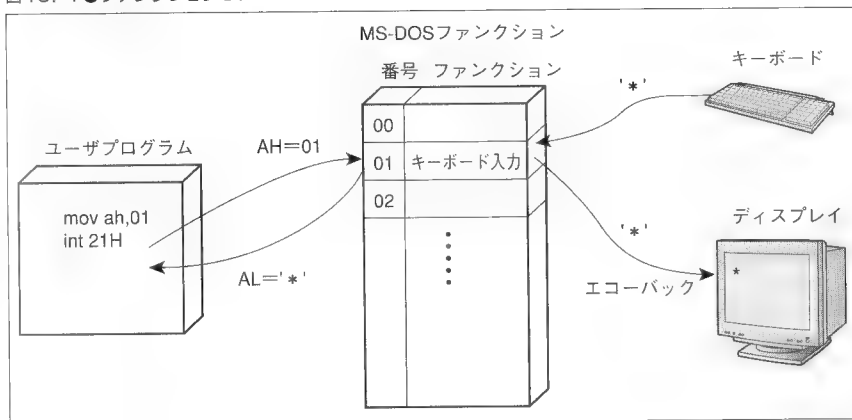
10.2 基本ファンクション

基本的な入出力装置からデータを入出力する例を示します.

■キーボードから1文字入力(エコーバック付き)

```
mov ah,1        ;ファンクション番号=1
int 21h         ;ソフトウェア割り込みでMS-DOSシステムコール
                ;ALレジスタに入力文字. ディスプレイ表示される.
```

キーボードから1文字入力するファンクションで, AHレジスタにファンクション番号を設定します. **図10. 1**に示すようにユーザプログラムがMS-DOSファンクション01を呼び出し, キーボード入力があるまで入力待ちを行います. たとえば文字*が入力されると, これがALレジスタに入ってユーザプログラムに戻されます. この文字*は自動的にエコーバックされて, ディスプレイに表示されます.

図10. 1●ファンクション01

■ディスプレイから1文字出力

```
mov ah,2            ;ファンクション番号=2
mov dl,'*'          ;出力する文字をDLレジスタへ設定
int 21h             ;ソフトウェア割り込みでMS-DOSシステムコール
```

ディスプレイへ1文字出力するファンクションで，AHレジスタにファンクション番号，DLレジスタへ出力文字を設定します．ユーザプログラムからMS-DOSファンクション02を呼び出し，ディスプレイに表示されます．

■キーボードから1文字入力(エコーバックなし)

```
mov ah,8            ;ファンクション番号=8
int 21h             ;ソフトウェア割り込みでMS-DOSシステムコール
                    ;ALレジスタに入力文字．ディスプレイ表示される．
```

キーボードから1文字入力するファンクションで，AHレジスタにファンクション番号を設定します．ユーザプログラムがMS-DOSファンクション08を呼び出し，キーボード入力があるまで入力待ちを行います．たとえば文字*がキー入力されると，ALレジスタによってユーザプログラムへ返されます．この文字*はディスプレイには表示されません．

■ディスプレイへ文字列を出力

```
    mov ah,9            ;ファンクション番号=9
    lea dx,strg         ;文字列アドレスをDXレジスタへ設定
    int 21h             ;ソフトウェア割り込みでMS-DOSシステムコール
strg db 'x86 Assembler','$'     ;表示文字列．文字列の最後は文字$
```

ディスプレイへ文字列をバッファから出力するファンクションで，AHレジス

タにファンクション番号を設定します．DXレジスタには，文字列のあるアドレスを設定するためlea命令を用います．ユーザプログラムがMS-DOSファンクション09を呼び出し，文字$が見つかるまで文字列を表示します．

■キーボードから文字列を入力

```
        mov ah,0aH          ;ファンクション番号=0A
        lea dx,strg         ;文字列アドレスをDXレジスタへ設定
        int 21h             ;ソフトウェア割り込みでMS-DOSシステムコール

    strg db 'x86 Assembler','$'      ;表示文字列．文字列の最後は文字$
```

キーボードから文字列をバッファへ入力するファンクションで，AHレジスタにファンクション番号を設定します．DXレジスタには，文字列のあるバッファアドレスを設定するためlea命令を用います．上の例ではstrgが文字列のバッファとなります．

■プロセスの終了

```
        mov ah,4c           ;ファンクション番号=4C
        mov al,0            ;リターンコード
        int 21h             ;ソフトウェア割り込みでMS-DOSシステムコール
                            ;ALレジスタに入力文字．ディスプレイ表示される．
```

呼び出しのあったプログラム(MS-DOSの定義ではプロセス)へ戻るファンクションで，AHレジスタにファンクション番号を設定します．ALレジスタにはリターンコードを設定します．ユーザプログラムでこのファンクションを呼び出すと，MS-DOSへ復帰することができます．

10.3 ファンクションのマクロ化

ここでは基本ファンクションをマクロ化します．これらマクロは4章以降で実際に使用されています．

リスト10．1にMS-DOSシステムコールを行うマクロ定義ファイルiomac.incを示します．1文字入力するkeyin，1文字を表示するdisp，MS-DOSへ復帰するexit，1行改行するcrlfなどを定義してあります．マクロは，

```
disp macro          ;マクロ名
     mov ah,2h      ;実際に展開される命令
     int 21h        ;実際に展開される命令
  endm              ;マクロの終了
```

の形式で記述します．マクロを使用する場合，このdispという名前を記述すると，アセンブラが中に記述された命令群をその場所に展開してくれます．プログラムの可読性がよくなります．マクロアセンブラMASMはマクロ記述ができるアセンブラを意味しています．マクロの詳細は9章『疑似命令とマクロ命令』を参照してください．

リスト10．1●MS-DOSシステムコール　マクロ定義（iomac.inc）

```
 1: CR   equ 0dh
 2: LF   equ 0ah
 3: ;--------------------------------
 4: keyin macro               ;キーボードから1文字入力(エコーバック). ALに入力文字
 5:      mov ah,1h            ;ファンクション1を指定
 6:      int 21h              ;MS-DOSのシステムコール(割込み)
 7:      endm
 8: ;--------------------------------
 9: keyin0 macro              ;キーボードから1文字入力(エコーバックなし). ALに入力文字
10:      mov ah,8h            ;ファンクション8を指定
11:      int 21h              ;MS-DOSのシステムコール(割込み)
12:      endm
13: ;--------------------------------
14: keyins macro             ;キーボードから文字列をバッファへ入力. DXにバッファアドレス
15:      mov ah,0ah           ;ファンクション0a指定
```

```
16:     int 21h                 ;MS-DOSのシステムコール(割込み)
17:     endm
18: ;-------------------------------
19: disp  macro                 ;ディスプレイに1文字出力. DLに出力文字
20:     mov ah,2h               ;ファンクション2を指定
21:     int 21h                 ;MS-DOSのシステムコール(割込み)
22:     endm
23: ;-------------------------------
24: disps macro                 ;ディスプレイに文字列を表示. DXに文字列アドレス
25:     mov ah,9h               ;ファンクション9を指定
26:     int 21h                 ;MS-DOSのシステムコール(割込み)
27:     endm
28: ;-------------------------------
29: print macro                 ;プリンタから1文字印刷. DLに印刷文字
30:     mov ah,5h               ;ファンクション5を指定
31:     int 21h                 ;MS-DOSのシステムコール(割込み)
32:     endm
33: ;-------------------------------
34: exit macro                  ;プロセスの終了して、MS-DOSへ復帰
35:     mov ax,4c00h            ;ファンクション4cを指定.リターンコード0
36:     int 21h                 ;MS-DOSのシステムコール(割込み)
37:     endm
37: ;-------------------------------
38: crlf  macro                 ;ディスプレイを1行改行
39:     mov dl,CR               ;CR(復帰)コード
40:     disp                    ;表示マクロ
41:     mov dl,LF               ;LF(改行)コード
42:     disp                    ;表示マクロ
43:     endm
44:  ;-------------------------------
```

演習問題 10

1. 次のプログラムを実行しなさい．
   (1) キーボードから1文字エコーバックなしで入力し，これをディスプレイ表示する．
   (2) キーボードから数字を1文字入力し，これを1バイトの数値に変換する．
   (3) キーボードから文字列を入力し，これをディスプレイ表示する．
   (4) 9章演習(1)で定義したマクロ"日付の読み出し"を用いて，日付を表示する．
   (5) 9章演習(2)で定義したマクロ"時刻の読み出し"を用いて，時刻を表示する．

第11章

基本プログラミング

本章では，8086アセンブラの基本プログラミングの例題を示します．アセンブラプログラムを用いたいターゲットのコンピュータはそれぞれシステムの構成が異なっています．ここではMS-DOSの上で，MS-DOSシステムコールを用いて，プログラムを実行することにします．これによりシステム構成の違いを吸収することができます．ここでの例題は，MS-DOSを理解しようとするものではなく，あくまでも8086アセンブラのプログラミングを理解しようとするものです．

アセンブリ言語でのプログラミングには，重要なことがあります．

①プログラムを機能ごとに分割して，サブルーチン化(あるいはライブラリ化)する．

②プログラムには，なるべく1行ごとにコメントを付ける．

アセンブラプログラムは1行に1命令を記述して行くため，プログラムの行数が非常に多くなり，プログラム全体の把握が難しくなります．そのために，プログラムを機能ごとに分割して，機能を把握しやすいプログラムにする必要があるのです．サブルーチン化したプログラムは，アセンブルしてオブジェクトファイル(.obj)にしておけば，これをリンクすることで，別のプログラムからも利用することができます．既に動作が検証されているサブルーチンを用いることで，プログラムの作成時間を短縮することができます．

またプログラムには，常にバグの危険性が存在します．どのような機能を求めてそのプログラムを記述したか，何を意図してその命令を用いたかなどを，コメントにすることによって，プログラムの理解やバグの発見を容易にすることができます．アセンブラプログラミングはやっかいなように見えますが，上記2項を守れば，以外に簡単にプログラミングができるものです．本章では6章4節で定義した基本サブルーチンを用いて，プログラミングを行います．

11.1 文字表示

ディスプレイに1文字表示する

基本サブルーチン群を用いて文字を表示するプログラムを**リスト11．1**に示します．行4:の.model smallはスモールモデルでプログラムを構成することを宣言する簡略化疑似命令です．スモールモデルはコード，データ，スタックセグメントがそれぞれ64kバイト以下で，実行形式としてXXX.exeファイルを生成します．

行6:の.dataはデータセグメント定義の簡略化疑似命令で，ここからデータセグメントが開始されることを宣言するものです．ここでは変数asterをdbによ

ってbyte変数であることを宣言し，その初期値として文字'*'を与えています．

行9:の.stack 100Hは，100Hバイトのスタック領域をとることを宣言しています．これ以前に.dataで宣言されていたデータ領域はここで終了となります．このプログラムでは，直接データをスタックに保存する命令はありませんが，サブルーチンを呼び出す際にリターンアドレスの保存にスタックが用いられるため，この宣言が必要になります．

行11:の.codeはコードセグメントがここから始まることの宣言です．dispcと名付けたメインプログラム(メイン手続き)がここから始まり，このプログラムが実行されるには，farプログラムとして呼び出されることを示しています．なお，ここで名付けられた名称dispcは，endp疑似命令でこの手続きを閉める際にもう一度用います．すなわち，手続きは同じ名称から同じ名称までが1かたまりとなります．慣れるまでは奇妙に感じるかもしれませんが，1つの手続きの範囲を明確にするために，このような方法が用いられています．このdispc番地からプログラムを実行するようにシステム(ここではMS-DOS)へ指示するため，行24:のend疑似命令には実行開始番地を示す，dispcが付随しています．行12:のextrnで，サブルーチンdispとexitが別ファイルにあることを宣言しています．

行15:～16:までが，実行時のDSレジスタを，data領域にするよう設定を行っています．@dataは.data簡略化疑似命令で宣言した場合の，データセグメントアドレスを示します．DSへ直接mov ds,@dataと書きたいところですが，DSへアドレスを直接転送するmov命令はないため，AXレジスタに一度書き込んで，AXレジスタからDSレジスタへアドレスを渡しています．

行18:はmov命令で，変数asterからDLへデータを転送しています．asterは文字'*'で初期化されているため，DLは文字'*'を持つことになります．次に基本サブルーチンdispを呼び出しています．これによりDLで渡した'*'文字が表示されます．

行21:のcall exit では，メインプログラムのdispcを終了し，MS-DOSへ復帰します．行22:はユーザプログラムのメイン手続きの終了を示すendp疑似命令です．dispc手続きの終了であることを示すため，名称dispcが再び用いられています．

行24:のend疑似命令は，プログラムの記述がここで終了していることと，付

随するdispcによってこのプログラムが，dispc番地から実行開始されることを処理系に伝えています．

実行例がプログラムの後ろに示してあります．プログラムを実行すると，単に'*'が表示されます．

リスト11．1●文字表示（簡略化セグメント定義）：disp.asm

```
 1: ;
 2: ; 文字表示
 3: ;
 4: .model small
 5: ;------------------------------------------------
 6: .data                      ;データセグメント
 7: aster db '*'               ;文字*
 8: ;------------------------------------------------
 9: .stack 100h                ;スタックセグメント領域100
10: ;------------------------------------------------
11: .code                      ;コードセグメント
12:     extrn disp:near,exit:near ;外部関数
13: ;-------------------------------
14: dispc proc far
15:     mov ax,@data           ;データセグメントのアドレスをaxへ
16:     mov ds,ax              ;dsレジスタへアドレスを設定
17: ;-------------------------------
18:     mov dl,aster           ;*文字をDLレジスタへ転送
19:     call disp              ;1文字表示サブルーチンコール
20: ;-------------------------------
21:     call exit              ;プロセス終了し、MS-DOSへ復帰
22: dispc endp
23: ;------------------------------------------------
24:     end dispc


実行例
    C:¥prg>disp
    *
```

リスト11．2に**リスト11．1**とまったく同じ働きをする，完全なセグメント定義をした文字表示プログラム例を示します．ここでは簡略化セグメント疑似命令ではなく，完全にセグメントを記述します．簡略化セグメント定義に比べ

ると多少複雑になりますが，MS-DOSに依存しないプログラムが必要な場合はこちらで記述するとよいでしょう．

行4:のassume疑似命令は，プログラム内のセグメントの構成をアセンブラ処理系に伝える働きをします．ここでは，mydataがデータセグメント，mystackがスタックセグメント，mycodeがコードセグメントであることを示しています．行5:のextrn疑似命令では，サブルーチンdispとexitが外部手続きで，さらにnearによってメモリアドレス上の距離が示されています．nearでは，dispとexitがこのプログラムと同一のセグメント内にあり，2バイトのアドレスでアクセスできることが示されます．

行7:～9:のmydata segment～mydata endsがデータセグメントで，変数asterが定義されています．行11:～14:までがスタックセグメントの定義でdw 100h dup(?)によって初期化しない100Hワードの領域をとります．bottom label wordは，スタックの底を示すためのラベルを定義し，後にスタックポインタSPの初期化のために使用します．

行16:のmycode segment publicはコードセグメントの始まりを示します．属性publicによって，同一のセグメント名mycodeが外部モジュールにある場合，これが連結されて1つの連続したセグメントとなります．外部にあるサブルーチンを使用して，それらをnearでアクセスする場合には，public属性を書く必要があります．

行18:～21:では，データセグメントとスタックセグメントのアドレスをそれぞれのセグメントレジスタへ設定します．行22:のlea sp,bottomでは，スタックポインタSPの初期値を設定します．簡略化セグメント定義を用いる場合はユーザがスタックポインタの設定を行う必要はありませんが，完全なセグメント定義を用いる場合には，この設定が必要となります．

行24:～の処理部分は簡略化セグメント定義の場合と変わりありません．

リスト11．2●文字表示（完全なセグメント定義）：dispc.asm

```
1: ;
2: ;        文字表示（完全なセグメント定義）
3: ;
4: assume ds:mydata,ss:mystack,cs:mycode
```

```
 5: extrn disp:near,exit:near     ;外部関数
 6: ;-------------------------------------------------
 7: mydata  segment             ;データセグメント
 8: aster db '*'                ;文字*
 9: mydata  ends
10: ;-------------------------------------------------
11: mystack segment stack       ;スタックセグメント
12:     dw 100h dup (?)         ;100Hワードの領域をとる
13: bottom  label word          ;スタックボトム
14: mystack ends                ;スタックセグメントの終了
15: ;-------------------------------------------------
16: mycode  segment public      ;コードセグメント
17: dispc   proc far            ;メインプログラム(far)
18:         mov ax,mydata       ;データセグメントのアドレス
19:         mov ds,ax           ;DSを設定
20:         mov ax,mystack      ;スタックセグメントのアドレス
21:         mov ss,ax           ;SSを設定
22:         lea sp,bottom       ;スタックポインタSPを設定
23: ;------------------------------
24:         mov dl,aster        ;*文字をDLレジスタへ転送
25:         call disp           ;1文字表示サブルーチンコール
26: ;------------------------------
27:         call exit           ;プロセス終了し，MS-DOSへ復帰
28:         dispc endp
29: mycode  ends
30: ;-------------------------------------------------
31:         end dispc
```

実行例

```
C:¥prg>dispc
*
C:¥>
```

11.2 キー入力，ディスプレイ表示

キーボードから1文字を入力し，これをディスプレイ表示する．

キーボードから文字を1文字入力し，これをディスプレイで表示するプログラムを**リスト11．3**に示します．データとスタックの設定は前例と変わりありま

せん．処理部分は行19:～23:の部分ですから，簡単に理解できるでしょう．行19:～20:で，文字入力を促すクエスチョンマーク(?)をdispサブルーチンを用いて表示します．次に基本サブルーチンkeyin0を用いて，エコーバックなしで1文字をキーボードから入力します．入力された1文字はDLレジスタへ書き込んで，dispを呼んで表示します．

実行例がプログラムリストの後に示してあります．この例では，英字'u'をキーインしてこれを表示しています．

リスト11．3●キー入力，ディスプレイ表示：i&o.asm

```
 1: ;
 2: ;キー入力，ディスプレイ表示
 3: ;
 4: .model  small
 5: ;-------------------------------------------------
 6: .data                       ;データセグメント
 7: qmark   db '?'              ;文字?
 8: ;-------------------------------------------------
 9: .stack 100h                 ;スタックセグメント領域100h
10: ;-------------------------------------------------
11: .code                       ;コードセグメント
12:     extrn keyin0:near,disp:near,exit:near      ;外部関数
13: ;--------------------------------
14: inout   proc far
15: ;--------------------------------
16:         mov ax,@data        ;データセグメントのアドレス
17:         mov ds,ax           ;dsレジスタへアドレスを設定
18: ;--------------------------------
19:         mov dl,qmark        ;入力促進文字?
20:         call disp           ;?を表示
21:         call keyin0         ;1文字キーイン
22:         mov dl,al           ;入力文字をDLへ
23:         call disp           ;1文字表示
24: ;--------------------------------
25:         call exit           ;プロセス終了し，MS-DOSへ復帰
26: inout   endp
27: ;-----------------------------------------------
28: end inout
```

```
実行例
    C:¥prg>i&o
    ?u
```

11.3 入出力の繰り返し

> キーボード入力とディスプレイ表示を繰り返すプログラム．繰り返しはEnterキーが押されたときに終了する．

リスト11．4にキーボード入力とディスプレイ表示を繰り返すプログラム例を示します．セグメントの設定などは前プログラムと同様ですが，行8:でEnterキーのコード0Dを定義しています．WindowsやMS-DOSシステムでは，文字コードとしてはASCIIコードが用いられます．コード0DはASCIIコードではCR(carriage return)コードで，Enterキーを押すとこのコードが入力されます．

実際の処理部分は行20:～28:で，最初に入力を促す「?」を表示します．入力と表示は前例と同様に基本サブルーチンkeyin0とdispを用いて行います．行24:のcmp al,enterでは，入力した文字とコード0Dを比較し，一致するなら次のje eopで，ラベルeopへジャンプしてプログラムを終了します．

入力文字がCRでないなら，その文字をdispサブルーチンを用いて表示します．次のjmp lpでは，繰り返しのためラベルlpへ無条件でジャンプします．

プログラムの後に，プログラムの実行例を示します．入力促進記号?の後のabcdefghijklmnがキーボードから入力された文字で，これを単純に表示したものです．

リスト11．4●入出力の繰り返し：rpi&o.asm

```
1: ;
2: ;キー入力，ディスプレイ表示繰り返し
3: ;
4: .model  small
5: ;-----------------------------------------------
6: .data                          ;データセグメント
```

```
 7: qmark   db '?'              ;文字?
 8: enter   db 0dh              ;Enterキーコード(CR)
 9: ;-------------------------------------------------
10: .stack 100h                 ;スタックセグメント領域100h
11: ;-------------------------------------------------
12: .code                       ;コードセグメント
13:     extrn keyin0:near,disp:near,exit:near ;外部関数
14: ;-------------------------------
15: repio   proc far
16: ;-------------------------------
17:         mov ax,@data        ;データセグメントのアドレス
18:         mov ds,ax           ;dsレジスタへアドレスを設定
19: ;-------------------------------
20:         mov dl,qmark        ;文字?
21:         call disp           ;?を表示
22: ;
23: lp:     call keyin0         ;1文字キーイン
24:         cmp al,enter        ;Enterキー入力?
25:         je eop              ;Enterなら終了へ
26:         mov dl,al           ;入力文字をDLへ
27:         call disp           ;1文字表示
28:         jmp lp              ;繰り返しのためlpへジャンプ
29: ;-------------------------------
30: eop:    call exit           ;プロセス終了し，MS-DOSへ復帰
31: repio   endp
32: ;-------------------------------------------------
33:         end repio
```

実行例

```
    C:¥prg>rpi&o
    ?abcdefghijklmn
```

11.4 文字列表示

あらかじめ用意した文字列をディスプレイに表示する．

文字列の表示は，1文字ずつの表示を繰り返すことで実現できます．しかし，MS-DOSには文字列を表示するファンクション09Hが用意されています．ここ

では簡単にするために，このファンクションを用いて表示することにします．MS-DOSの文字列には，最後に$(ダラーマーク)が付いている約束になっています．**リスト11．5**に文字列の表示プログラム例を示します．

行6:～7:でデータセグメントの初期化を行っています．

```
        strg        db 'Hello ,x86アセンブラ!',0dh,0ah,'$'
```

で，データはdbによってバイト単位で文字として初期化されます．ここで文字として表現できないASCIIコードCR(キャリッジリターン)とLF(ラインフィード)が，16進数で表現されています．0dhはCRのコードで，0ahはLFのコードです．通常CRはディスプレイのカーソルを行の先頭へ戻し，LFはカーソルを現在の表示位置の次の行に送る働きをします．CRとLFがディスプレイに渡されると，文字として表示されるのではなく，表示のための制御コードとして用いられます．

スタックセグメントの設定は前例と同様です．このプログラムでは文字列の表示に，MS-DOSシステムを用いて文字列を表示する基本サブルーチンdispsを用います．行19:ではデータセグメントに用意した文字列strgのアドレスをDXへ設定します．ここではlea命令を用いて文字列アドレスを設定しています．しかし，以下のように，

①lea dx,strg

②mov dx,offset strg

のいずれを用いても，アドレスの設定を行うことができます．②の場合は，offset strgとすることによって，DXレジスタへはstrg内のデータではなく，strgが置かれるメモリアドレスが渡されます．

行20:のdispsの呼び出しによって，指定したアドレスstrgから始めるデータ領域の文字を，$に行き着くまで表示します．もし$の記入を忘れると，メモリの内容を$と同じ内部コードに達するまで表示することになります．

行22:のexitの呼び出しでMS-DOSへ復帰します．

このプログラムを実行すると，実行例のようにディスプレイ上に表示されます．

リスト11．5●文字列表示プログラム：disps.asm

```
 1: ;
 2: ;     文字列表示
 3: ;
 4: .model small
 5: ;----------------------------------------
 6: .data                   ;データセグメント
 7: strg   db 'Hello ,x86アセンブラ!',0dh,0ah,'$'
                            ;文字列，CRとLF;$はデータ終了印
 8: ;----------------------------------------
 9: .stack 100h             ;スタックセグメント
10: ;----------------------------------------
11: .code                   ;コードセグメント
12: extrn disps:near,crlf:near,exit:near  ;外部関数
13: ;-------------------------------
14: kdisps proc far         ;メインプログラム
15: ;-------------------------------
16:         mov ax,@data    ;データセグメントのアドレス
17:         mov ds,ax       ;dsレジスタへアドレスを設定
18: ;-------------------------------
19:         lea dx,strg     ;文字列番地ををdxレジスタへ
20:         call disps      ;文字列表示プログラムの呼び出し
21: ;--------------------------
22:         call exit       ;プロセス終了し，MS-DOSへ復帰
23:         kdisps endp
24: ;-------------------------------
25:         end kdisps      ;dispsからプログラムを開始する
```

実行例

```
    C:¥prg>disps
    Hello ,x86アセンブラ!
```


11.5 文字列をカラー表示

あらかじめ用意した文字列をカウントし，ディスプレイにカラー表示する．

ここでも文字列の表示には，MS-DOSファンクション09Hを用います．MS-DOSの文字列には，最後に$(ダラーマーク)が付いている約束になっています．

文字をカラーにするために，エスケープシーケンスを用います．エスケープシーケンスはディスプレイを制御するため，ASCIIコードのESC(1BH)で始まる文字列をディスプレイに出力するものです．たとえば，カーソルの位置を移動したり，表示する文字の属性を変えてカラー表示することなどができます．**表11．1**にエスケープシーケンスの表示文字変更のエスケープシーケンスを示します．

表11．1●エスケープシーケンス

エスケープシーケンス	機　能
ESC[30m	表示文字の属性を、黒に設定する。
ESC[31m	赤
ESC[32m	緑
ESC[33m	黄
ESC[34m	青
ESC[35m	紫
ESC[36m	水色
ESC[37m	白

リスト11．6に文字列をカラーで表示プログラム例を示します．

行6:～13:でデータの定義と初期化を行っています．行7:ではエスケープコードESCcdが1BHであることを定義しています．stringは青～白の漢字を各色で表示するようにエスケープシーケンスを設定しています．

行22:～24:がセグメントレジスタの設定ですが，ストリング命令を用いるためESレジスタの設定も行っています．行26:～34:では文字列の文字数をカウントします．

行41:のdispsの呼び出しによって，指定したアドレスstrgから始めるデータ領域の文字を，$に行き着くまで表示します．もし$の記入を忘れると，メモリの内容を$と同じ内部コードに達するまで表示することになります．

行44:のexitの呼び出しでMS-DOSへ復帰します．

このプログラムを実行すると，実行例のようにディスプレイ上に表示されます．

リスト11．6●文字列をカラー表示：colst.asm

```
 1: ;
 2: ;   色のついた文字列の表示
 3: ;
```

```
 4: .model  small
 5: ;-----------------------------------------------
 6: .data  ;データセグメント
 7: ESCcd  equ 1bH          ;エスケープコード
 8: string db  ESCcd, '[34m青', ESCcd, '[31m赤'
 9:        db  ESCcd, '[35m紫', ESCcd, '[32m緑'
10:        db  ESCcd, '[36m水', ESCcd, '[33m黄'
11:        db  ESCcd, '[37m白$'
12: dolmark db  '$'         ;ダラーマーク
13: n      dw  ?            ;カウンタ
14: ;-----------------------------------------------
15: .stack 100H             ;スタックセグメント領域100H
16: ;-----------------------------------------------
17: .code                   ;コードセグメント
18:   extrn keyin:near,disp:near,crlf:near,exit:near      ;外部関数
19: ;------------------------------
20: colorstr proc far       ;main手続き
21: ;--------------------------
22:        mov ax,@data     ;データセグメントのアドレスをaxへ
23:        mov ds,ax        ;DSレジスタへアドレスを設定
24:        mov es,ax        ;ESレジスタへアドレスを設定
25: ;-------------------------
26:        clc              ;キャリフラグをクリヤ
27:        cld              ;方向を増加に
28:        lea si,string    ;文字列アドレス設定
29:        xor cx,cx        ;カウンタCXのクリア
30: cmplp: lodsb            ;文字列をロード，その後アドレスを1増
31:        cmp al,dolmark   ;ダラーマークと比較
32:        je brk           ;一致するならbrkへ
33:        inc cx           ;カウンタCXを1増
34:        jmp cmplp        ;一致しなければループ
35: brk:
36: ;
37:        cld              ;方向を増加に
38:        lea si,string    ;文字列アドレス設定
39: dsplp: lodsb            ;文字列をロード，SIを1増
40:        mov dl,al        ;文字をDLレジスタへ転送
41:        call disp        ;1文字表示サブルーチンコール
42:        loop dsplp       ;文字数だけループ
43: ;-------------------------
44:        call exit        ;プロセス終了し，MS-DOSへ復帰
45: colorstr   endp
46: ;-------------------------------------------------------------
47:        end colorstr     ;プログラムの終了.decadd番地から実行
```

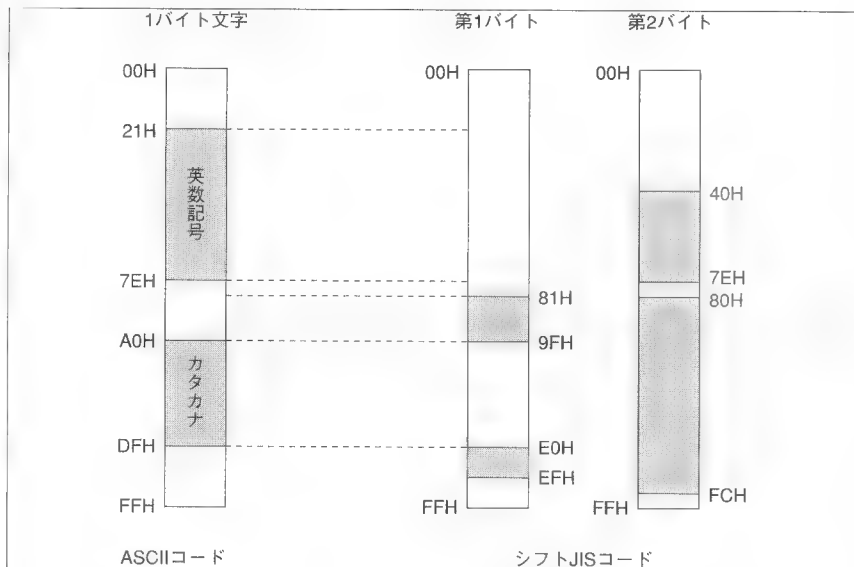
実行例

```
C:¥prg>colst
青赤紫緑水黄白
```

11.6 漢字の取り扱い

漢字をキーボード入力し，これをディスプレイ表示することを繰り返す．ESCキーが押されたとき入出力を終了する．

漢字は英字，数字，記号などASCIIコードで表現される文字とは異なった扱いとなります．ASCIIコードはコンピュータ内部では通常1バイトのデータとして取り扱われます．これに対して漢字は2バイトで1文字を表現します．漢字を表現するコードとしては，JIS漢字コード(JIS情報交換用漢字符号系C6226)があります．ところが，2バイトのJIS漢字コードの第1バイトと，ASCIIコードとが重なる部分があり，そのままではASCIIコードとJIS漢字コードを混在して使用することができません．そこでWindowsやMS-DOSでは，JIS漢字コード全体をコード表上で別の領域へシフトして，漢字コードの最初の1バイトがASCIIコードと重ならないようにした，通称シフトJISコードが用いられています．シフトJISコードは，**図11．1**に示すように，最初の1バイトをASCIIコードやJISコードが使用していない領域(81H～9FH,E0H～EFH)へシフトして，1バイトの英数字コードと2バイトの漢字コードが混在しても，最初の1バイトによって英数字か漢字かを識別可能にしたものです．

図11．1●ASCIIコードとシフトJISコード

漢字コードは2バイトで1文字が表現されるため，たとえばキーボードから漢字を入力するには第1バイトと第2バイトを順に1バイトずつ入力する必要があります．ディスプレイ表示の場合も同様です．

リスト11．7に漢字を入出力するプログラムを示します．行6:〜8:では，データセグメントに漢字1文字を保存するため，その第1バイト用にk1，第2バイト用にk2のバッファを用意します．スタックは100Hバイトを用意します．

行20:〜26:が漢字の入力部分です．最初にCXレジスタに漢字の最大数40を設定します．これで入力できる漢字は40文字となります．次にサブルーチンkeyin0を呼び出して，漢字の第1バイトを入力し，バッファk1へ保存します．keyin0を用いているため，エコーバックはありません．次に入力した第1バイトがESCコードかを比較します．ESCコードでないならラベルk2iへ飛び，第2バイトの入力に移ります．ESCコードならばcall exitでプログラムを終了します．漢字の第2バイトの入力とk2への保存は，行28:のcall keyin0と行29:のmov k2,alで行います．

漢字の表示は，行31:〜34:で行われます．単にmov命令と1文字表示サブルーチンdispを2度行うことで実現します．loop k1iでは，カウンタとして用い

ているCXレジスタを1減して，0でなければラベルk1iへ飛んで入出力を繰り返します．

プログラムの実行例は，' 漢字の入出力' と入力した後，ESCキーを押して終了したものです．

リスト11．7●漢字の取り扱い：ki&o.asm

```
 1: ;
 2: ;       漢字の取り扱い
 3: ;
 4: .model small
 5: ;---------------------------------------------
 6: .data                        ;データセグメント
 7: k1      db ?                 ;漢字バッファ1
 8: k2      db ?                 ;漢字バッファ2
 9: ;---------------------------------------------
10: .stack 100h                  ;スタックセグメント
11: ;---------------------------------------------
12: .code                        ;コードセグメント
13:   extrn keyin:near,disp:near,exit:near ;外部関数
14: ;------------------------------
15: kinout proc far              ;メインプログラム
16: ;------------------------------
17:         mov ax,@data         ;データセグメントアドレス
18:         mov ds,ax            ;データセグメントの設定
19: ;------------------------------
20:         mov cx,40            ;漢字入力を40文字までとする
21: k1i:    call keyin0          ;漢字の1byte目を入力(エコーなし)
22:         mov k1,al            ;漢字の1byte目を保存
23: ;
24:         cmp al,1BH           ;ESCキーがキーインされたか？
25:         jne k2i              ;ESCでないなら次へ
26:         call exit            ;ESCならプログラム終了
27: ;
28: k2i:    call keyin0          ;漢字の2byte目を入力(エコーなし)
29:         mov k2,al            ;漢字の2byte目を保存
30: ;
31:         mov dl,k1            ;漢字の1byte目
32:         call disp            ;表示
33:         mov dl,k2            ;漢字の2byte目
34:         call disp            ;表示
35: ;
36:         loop k1i             ;漢字40文字以下なら繰り返し
```

```
37: kinout  endp
38: ;----------------------------------------------
39:         end kinout          ;kinoutからプログラムを開始することを示す
```

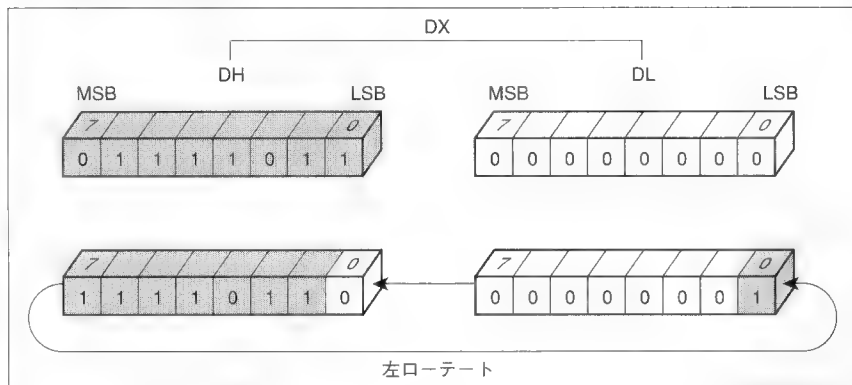
実行例

```
C:¥prg>ki&o
漢字の入出力
```

11.7 2進数をディスプレイに表示

あらかじめ用意した2進コード1バイトを，2進数字でディスプレイ表示する．2進コードは1バイトで，正の整数とする．

内部コードの2進数をディスプレイ上に表示するプログラム例を**リスト11．8**に示します．ここではbinと名付けた変数が，数値123で初期化されています．この数値を2進数として表示します．binはバイト変数なので8桁を表示することになります．この例では2進数をDHレジスタへ転送し，これをDLレジスタを含めて左ローテートすると，DHレジスタのMSBがDLレジスタのLSBへ入ります．この1桁分の表示を8桁繰り返すことで1バイトのデータを2進数で表示します．

図11．2●DXレジスタの状態

行4:でモデルをsmallとし，行15:～16:でデータセグメントのアドレスをDSレジスタへ設定しています．

行18:ではCXレジスタをカウンタとして使用するため8を与えます．行19:では変数binのデータをDHレジスタへ転送しています．行21:のラベルdspはこの行のxor命令のアドレスを示すために用います．ここではDLレジスタをクリアして，行22:のrol命令でDHレジスタのMSBを受け入れる用意をします．このrol命令でDLレジスタへローテートされる値は，0または1のいずれかです．この命令を繰り返すことで，2進数のMSBから1ビットずつDLレジスタへ移していきます．

一般に数字やアルファベットの内部コードは連続した値が与えられています．数字'1'の内部コードは，'0'の内部コードより1だけ大きい数値が割り当てられています．たとえば数値やアルファベットを表現するのに一般に用いられるASCIIコードでは，数字'0'は30H，'1'は31H，'9'は39Hが割り当てられています．したがって，数値1を数字'1'に変換したいときには，これに'0'のコードを加算してやれば，ちょうど'1'のコードが得られます．

行23:のadd命令では，1ビットのデータに数字'0'のコードを加算することで，'0'か'1'を作り出しています．行24:で文字表示サブルーチンdispを呼び出して，2進数の1桁を表示します．

行25:のloop命令が実行されると，自動的にCXレジスタの値がディクリメントされ，その値が0でなければdsp番地へジャンプします．CXへ与えられた初期値は8ですから，このループは8回繰り返されることになります．

行30:のcall exitでプログラムを終了し，MS-DOSへ復帰します．行31:endp疑似命令は，main手続きの最後を示します．

行33:はプログラム記述の終了を示すend疑似命令で，加えられたdispbinによって，このプログラムはdispbin番地から実行を開始することになります．

このプログラムを実行すると実行例のようにディスプレイ上に表示されます．

リスト11．8●2進コード→2進表示：b2dsp.asm

```
 1: ;
 2: ;2進コードを表示(8桁)
 3: ;
 4: .model  small
 5: ;-----------------------------------------------
 6: .data                       ;データセグメント
 7: bin     db 01111011B        ;2進数(10進 123)表示データ
 8: ;-----------------------------------------------
 9: .stack  100h                ;スタックセグメント領域100h
10: ;-----------------------------------------------
11: .code                       ;コードセグメント
12:   extrn disp:near,exit:near   ;外部関数
13: ;--------------------------------
14: dispbin proc far            ;main手続き
15:         mov ax,@data        ;データセグメントのアドレスをaxへ
16:         mov ds,ax           ;dsレジスタへアドレスを設定
17: ;--------------------------
18:         mov cx,8            ;カウンタを8に設定
19:         mov dh,bin          ;データをdhへ転送
20: ;
21: dsp:    xor dl,dl           ;dlをクリア
22:         rol dx,1            ;dxを左方向へローテート.1桁分がdlへ入る
23:         add dl,'0'          ;数字に変換するため,dlの数値に文字0を加算
24:         call disp           ;2進1桁を表示
25:         loop dsp            ;(cx)←(cx)-1 cxが0までdspから繰り返す
26: ;
27:         mov dl,'B'          ;2進数の最後のB
28:         call disp           ;Bを表示
29: ;--------------------------
30:         call exit
31: dispbin endp                ;手続きmainの終了
32: ;------------------------------------------------------------
33:         end dispbin         ;プログラムの終了. dispbin番地から実行
```

実行例

```
C:¥prg>b2dsp
01111011B
```

11.8 10進表示

あらかじめ用意した2進コード1バイトを，10進数字でディスプレイ表示する．

2進コードを10進数字に変換して，表示する例を**リスト11．9**に示します．実際は内部コードの2進数をBCDコードに変換し，これを10進数字にして表示します．2進コードは1バイトで，正の整数とします．正の整数とすることで，問題がかなり簡単になります．1バイトのデータが表現できる数値は0～255ですから，表示する10進桁数は最大3桁用意すればよいことになります．

行1:～6:までは前例題と同様で，行4:でモデルをsmallとしています．行6:でdigit equ 5としているのは，digitという名前が出てきた場合は，これを数字の5と見なせという疑似命令です．

行10:では変換する2進数binに01111011B(10進で123)を，除算に用いるため変数n10に10を初期値として与えます．

行12:の配列bcdはdigit桁だけのバイト数を用意し，これにBCDデータとして各桁を格納します．なお，dup(?)では初期化されないことを意味しています．

行21:～22:でデータセグメントのアドレスをDSレジスタへ設定しています．

行25:～28:では，確認のため2進コードを2進数字で表示します．まず2進コードbinをALレジスタへ転送して，サブルーチンdispbinを用いて2進表示します．dispbinは前例題で処理した2進コードを2進表示する部分を，サブルーチン化したものです．この後，変換を示すため記号>を表示します．

行31:～38:では2進コードを2進化10進数(BCD)各桁へ分解し，配列として用意したbcdへ各桁を保存します．行31:ではCXレジスタへBCDの桁数を設定していますが，これは2進コードをBCDへ分解する際のループのカウンタとして用います．行32:では10進数へ変換する2進コードを，ALレジスタへ転送しています．行33:ではインデックスレジスタDIへ，BCDを保存する最下位桁のメモリアドレスを渡しています．

行34:～38:では，実際に2進数をBCD桁に分解しています．xor ah,ahでAH

レジスタをクリアして，AHとALを連結した16ビットのデータとします．次のdiv n10はこのAH:ALの値を数値10で割る演算命令です．8086の除算命令では定数を用いてdiv 10の形をとることが許されていないため，n10と名付けたメモリ番地に数値10をおいています．このdiv命令では除算結果はALレジスタに，余りがAHレジスタへ入ります．AHに入る除算の余りは，BCDの下位桁となるのでこの値を，mov [di],ahでDIレジスタを用いた間接アドレスで保存します．最初のループでは，BCDの最下位桁が保存されます．

行37:では次の上位BCD桁の保存場所を用意するため，DIレジスタ内のアドレスをディクリメントしています．行38:のloop命令では，CXレジスタを自動的にディクリメントし，この値が0でなければcol番地へジャンプします．ループが終了するとbcdで始まる5バイトのデータ領域に，binの値をBCDに変換した値が入ります．

行41:～47:ではBCDの各桁を，10進数として表示します．まずmov cx,digitでCXレジスタにBCDの桁数を設定して，ループカウンタとします．行42:ではSIインデックスレジスタへBCDのメモリアドレスを渡しています．行43:ではSIレジスタによって間接アドレスでBCDを上位桁から1桁ずつDLレジスタへ送り，行44:でこれに数字0の内部コードを加算しています．これによりBCDの数値に対応した0～9までの数字のいずれかが得られます．次にdispサブルーチンを呼び出して，1桁のBCDを10進数として表示します．行46:ではBCDアドレスをインクリメントして次の桁を参照する準備を行い，次のloop命令でCXを自動的にディクリメントして，0でなければdsp番地へ戻って，BCD下位桁を表示することを繰り返し実行します．

このプログラムを実行すると，実行例のようにディスプレイ上に表示されます．

リスト11．9●2進コード→10進表示：b10dsp.asm

```
1: ;
2: ;2進コードを10進表示
3: ;
4: .model  small
5: ;
6: digit   equ 3
7: ;-----------------------------------------------
```

```
 8: .data                            ;データセグメント
 9: ;
10: bin     db  01111011B            ;2進数(10進 123)
11: n10     db  10                   ;数値10
12: bcd     db  digit dup(?)         ;10進数を桁数だけ用意
13: ;-------------------------------------------------
14: .stack 100h                      ;スタックセグメント領域100h
15: ;-------------------------------------------------
16: .code                            ;コードセグメント
17:   extrn disp:near,exit:near    ;外部関数
18: ;----------------------------
19: dispdec proc far                 ;main手続き
20: ;--------------------------
21:         mov ax,@data             ;データセグメントのアドレスをaxへ
22:         mov ds,ax                ;dsレジスタへアドレスを設定
23: ;--------------------------
24: ;2進コードを2進表示
25:         mov al,bin               ;2進数をalへ
26:         call dispbin             ;2進表示
27:         mov dl,'>'               ;変換表示のため
28:         call disp                ;>を表示
29: ;--------------------------
30: ;2進コードをBCD桁に分解
31:         mov cx,digit             ;BCD分解のため，カウンタに桁数を設定
32:         mov al,bin               ;2進コードをalへ転送
33:         lea di,bcd+digit-1       ;BCDの保存アドレス
34: col:    xor ah,ah                ;ahをクリア
35:         div n10                  ;(ah:al)/10 余りはBCD下位桁
36:         mov [di],ah              ;余りをBCDとして保存
37:         dec di                   ;BCD保存アドレスを1減．上位桁へ
38:         loop col                 ;cxを1減．0でなければcolに飛び繰り返す
39: ;--------------------------
40: ;BCD桁を10進表示
41:         mov cx,digit             ;BCD桁数をカウンタに設定
42:         lea si,bcd               ;BCDの保存アドレスを設定
43: dsp:    mov dl,[si]              ;BCD1桁をDLにロード
44:         add dl,'0'               ;BCDの値に数字0を加算→数字化
45:         call disp                ;BCDの1桁を10進表示
46:         inc si                   ;BCD保存アドレスを1増
47:         loop dsp                 ;cxを1減.0でなければdspに飛び繰り返す
48: ;
49:         mov dl,'D'               ;10進数の最後のD
50:         call disp                ;Dを表示
51: ;--------------------------
```

```
52:         call exit          ;プロセス終了し，MS-DOSへ復帰
53:         dispdec endp       ;手続きmainの終了
54: ;-------------------------
55: dispbin proc near          ;1バイトの2進コードを10進数字で表示.2進数はalで受ける
56:         mov cx,8           ;カウンタを8に設定
57:         mov dh,al          ;データをdhへ転送
58: ;
59: dsp2:   xor dl,dl          ;dlをクリア
60:         rol dx,1           ;dxを左方向へローテート.1桁分がdlへ入る
61:         add dl,'0'         ;数字に変換するため，dlの数値に文字0を加算
62:         call disp          ;2進1桁を表示
63:         loop dsp2          ;(cx)←(cx)-1 cxが0までdsp2からを繰り返す
64: ;
65:         mov dl,'B'         ;2進数の最後のB
66:         call disp          ;Bを表示
67:         ret
68: dispbin endp
69: ;------------------------------------------------------------
70:         end dispdec        ;プログラムの終了.dispdec番地から実行
```

実行例

```
C:¥prg>b10dsp
01111011B>123D
```

11.9 16進表示

あらかじめ用意した2進コード1バイトを，16進数字でディスプレイ表示する.

2進コードを16進数字に変換して表示する例を，リスト11. 10に示します.内部コードの2進数を上下4ビット(nibble)に分割し，これをそれぞれ16進数字にして表示します．2進数は1バイトで，表現できる数値は0～255ですから，表示する16進桁数は2桁用意すればよいことになります.

行1:～6:までは前例題と同様で，行7:～13:までがデータの定義になります．binは表示する2進数で，初期値として01111011B(10進で123)を与えます.

htableは4ビットの2進コードを16進数字に変換するためのテーブルで，16進数字'0123456789ABCDEF'を初期値として与えます．hexhとhexlは，1バイトのデータを4ビットに分解し，上下4ビットを保存するための変数で，?によって初期化しないことを意味します．

行14:～22:では，前例題と同様にスタックの設定，外部関数の宣言，データセグメントのアドレスを，DSレジスタへ設定しています．

行25:～28:では，確認のため2進コードを2進数字で表示します．まず2進コードbinをALレジスタへ転送して，サブルーチンdispbinを用いて2進表示します．この後，変換を示すため記号>を表示します．

行31:～39:では，2進コードを上下4ビット(nibble)に分解し，各桁をhexhとhexlへ保存します．行31:では10進数へ変換する2進コードbinを，ALレジスタへ転送しています．次のand al,11110000Bで，下位4ビットをマスクし上位4ビットを抽出します．抽出した4ビットを右シフトして，4ビットの2進数として扱えるようにし，これを変数hexhへ保存します．

1バイトの2進数の分解は10進数の分解と同様に，除算を用いて行うこともできます．その例を下に示します．

```
.data
n10     db 10
.code
;1byteの2進数を,4bitずつに分解(除算を用いる)
        mov al,bin      ;データをALへ転送
        xor ah,ah       ;ahをクリア
        div n16         ;(AH:AL)/16 余りは16進下位桁
        mov hexl,ah     ;余りを下位桁として保存
        mov hexh,al     ;商を上位桁として保存
```

この分解法の方がプログラムとしてはわかりやすいかもしれません．しかし除算を用いる方が，プログラム例のマスクを用いる場合より実行時間がかかります．アセンブリ言語プログラムでは，実行時間がより短い方法をとることが望

ましい場合が多いので，マスクを用いる方がよいでしょう．

行42:〜54:では4ビットの2進数を，16進数として表示します．まず16進数字のテーブルhtableをBXレジスタへ設定して，xlat命令のテーブルアドレスとします．

行43:〜46:では上位4ビットの変換と表示を行います．まずhexhから4ビットの2進数をALレジスタへ設定し，次のxlat命令によって，ALの数値に対応する16進数字に変換します．変換した数字を基本サブルーチンdispによって，表示することになります．

行48:〜51:では，下位4ビットの変換と表示を行います．上位4ビットと同様にhexlをALに設定し，xlat命令で対応する16進数字に変換，dispによって表示を行います．上下4ビットの16進数を表示して，1バイト分の2進データを表示したことになります．

行53:〜54:では，数字の最後にHを表示して，16進数であることを示します．このプログラムを実行すると，実行例のようにディスプレイ上に表示されます．

リスト11．10●2進コード→16進表示：b16dsp.asm

```
 1: ;
 2: ;2進コードを16進表示
 3: ;
 4: .model  small
 5: ;
 6: ;-----------------------------------------------
 7: .data                          ;データセグメント
 8: ;
 9: bin     db  01111011B          ;2進数(10進 123)
10: htable   db  '0123456789ABCDEF'
11: hexh    db  ?              ;16進数の上位4bit
12: hexl    db  ?              ;         下位4bit
13: ;-----------------------------------------------
14: .stack 100h                ;スタックセグメント領域100h
15: ;-----------------------------------------------
16: .code                      ;コードセグメント
17: extrn disp:near,exit:near ;外部関数
18: ;--------------------------------
19: disp16 proc far            ;mainプログラム
```

```
20: ;---------------------------
21:         mov ax,@data       ;データセグメントのアドレスをaxへ
22:         mov ds,ax          ;dsレジスタへアドレスを設定
23: ;---------------------------
24: ;変換前の2進数を表示
25:         mov al,bin         ;1byteの2進数をalへ
26:         call dispbin       ;2進表示
27:         mov dl,'>'         ;変換表示のため
28:         call disp          ;>を表示
29: ;---------------------------
30: ;1byteの2進数を,4bitずつに分解
31:         mov al,bin         ;1byteの2進数をalへ
32:         and al,11110000B   ;上位4ビットを抽出
33:         mov cl,4           ;カウンタCL
34:         shr al,cl          ;4ビット右へシフト
35:         mov hexh,al        ;上位桁として保存
36: ;
37:         mov al,bin         ;1byteの2進数をalへ
38:         and al,00001111B   ;下位4ビットを抽出
39:         mov hexl,al        ;下位桁として保存
40: ;---------------------------
41: ;4bitを16進表示
42:         lea  bx,htable     ;16進文字テーブルの先頭番地を設定
43:         mov  al,hexh       ;上位4bitを設定
44:         xlat               ;bxが指すテーブルの(al)番目の項目をalへ
45:         mov  dl,al         ;表示のためデータをdlへ
46:         call disp          ;1文字表示
47: ;
48:         mov  al,hexl       ;下位4bitを設定
49:         xlat               ;bxが指すテーブルの(al)番目の項目をalへ
50:         mov  dl,al         ;表示のためデータをdlへ
51:         call disp          ;1文字表示
52: ;
53:         mov dl,'H'         ;16進数の最後にHを表示
54:         call disp          ;Bを表示
55: ;---------------------------
56:         call exit          ;プロセス終了し,MS-DOSへ復帰
57: disp16  endp               ;手続きmainの終了
58: ;---------------------------
59: ;1バイトの2進コードを2進数字で表示
60: ;
61: dispbin proc near          ;コードはalで受ける
62:         mov cx,8           ;カウンタを8に設定
63:         mov dh,al          ;データをdhへ転送
```

```
64: ;
65: dsp2:   xor dl,dl          ;dlをクリア
66:         rol dx,1           ;dxを左方向へローテート.1桁分がdlへ入る
67:         add dl,'0'         ;数字に変換するため,dlの数値に文字0を加算
68:         call disp          ;2進1桁を表示
69:         loop dsp2          ;(cx)←(cx)-1 cxが0までdsp2からを繰り返す
70: ;
71:         mov dl,'B'         ;2進数の最後にBを表示
72:         call disp          ;Bを表示
73:         ret
74: dispbin endp
75: ;-----------------------------------------------------------------
76:         end disp16         ;プログラムの終了.disp16から実行
```

実行例

```
C:¥prg>b16dsp
01111011B>7BH
```

11.10 10進入力を2進コードに変換

キーボードから10進数入力し，これを2進コードに変換，表示する．10進数は4桁以内とする．

キーボードから4桁以下の10進数を入力して，各桁を2進化10進数に変換した後，2進コードに変換して表示するプログラム例を**リスト11．11**に示します．

行1:～6:までは前例題と同様で，行7:～13:までがデータの定義になります．変数dgtmaxは10進数の最大桁数の定義，digitは実際に入力された桁数を保持します．binは10進数を最終的に2進数に変換した値を保持します．配列bcdは，入力された10進各桁を2進化10進数(BCD)に変換して保持します．

行15:～23:では，前例題と同様にスタックの設定，外部関数の宣言，データセグメントのアドレスを，DSレジスタへ設定しています．

行26:～36:では10進数を入力し，各桁をBCDへ変換して配列bcdへ保存します．まず，レジスタCXを入力桁数のカウンタとして用いるため，最大桁数

dgtmaxを設定します．次にlea si,bcdで，レジスタSIにBCD各桁の保存配列bcdのアドレスを設定します．

行31:～35:が実際に10進数を入力する部分で，ループを構成しています．10進各桁の入力にはkeyinサブルーチンを用います．10進数字からBCDへの変換は，数字－'0'を計算することで行い，SIレジスタを用いた間接アドレシングでbcdへ保存します．

WindowsやMS-DOS系コンピュータ内部では，数字はASCIIコードで表現されます．数字0～9は連続したコードとして定義されており，これを**表11．2**に示します．数字'0'のコードは30Hで，'9'のコードは39Hとなっており，その差は9となっています．したがって，

数字－'0'

の計算をすることで，10進1桁分の2進コード(BCD)に変換することができるわけです．

表11．2●数字コード

数字	ASCIIコード	HEX
0	0011　0000	30
1	0011　0001	31
2	0011　0010	32
3	0011　0011	33
4	0011　0100	34
5	0011　0101	35
6	0011　0110	36
7	0011　0111	37
8	0011　1000	38
9	0011　1001	39

行39:～40:では実際に入力された桁数を求めます．数字入力の際，CXの初期値は最大桁数でしたから，ループごとに1減され，ループ終了時にCXは[最大桁数－入力桁数]となっています．そこで[最大桁数－CX]の計算を行って，実際の入力桁数を求め，その値はdigitに保存されます．

行43:～51:では，BCDに分解されたものを純2進数に変換します．たとえば10進数123は，

$$\mathbf{123 = 1 \times 10^2 + 2 \times 10^1 + 3}$$

と表現することができます．プログラム上の実際の変換方法はこの手法用いて，

上位BCD×10＋下位BCD

の計算を繰り返すことで行います．まずカウンタCXに実際の入力桁数を設定し，SIにBCDの先頭アドレスを設定します．計算途中で2進数を一時的に保持するAXと補助的に用いるBHレジスタをxor命令を用いてクリアします．

行47:～51:のループでは，AXにある2進コードに10をかけることによって，桁上げをします．第1回目のループでは，AXはクリアされているので結果は0となります．次にSIを用いた間接アドレスでBCDをBLにロードし，これをAXに加算します．BCDがロードされたBLはバイトですが，加算の際はBXを用います．BHレジスタがこのループに入る前に，クリアされているためこの計算が可能になります．BCDはループごとに，上位桁→下位桁の順でロードされます．第1回目のループではBCDの最上位桁がロードされることになります．次のループで下位のBCDをアクセスするため，inc siでレジスタSIを1増加します．

ループの終了はCXレジスタの値がゼロになったときに起こります．ループ終了後に，2進コードに変換した値を変数binに保存します．

行56:～67:では入力した10進数を，2進数として表示します．最初に変換を示すため記号>をdispを用いて表示します．サブルーチンdispbinは，ALレジスタの1バイトデータを表示するためのサブルーチンであるため，上位バイトと下位バイトの2回に分けて表示します．まず表示する2進コードbinをAXレジスタへロードし，xchg命令で上位バイトと下位バイト交換します．これでALレジスタには，上位バイトが入ったことになります．ここでdispbinを呼び出し上位バイトを表示します．

下位バイトの表示は，AXレジスタに2進コードをロードし，ALにある下位バイトをdispbinを用いて表示します．この後，2進数を表現するBを表示します．

この10進入力では，本当に数字が入力されたかの確認をしていません．必要ならば次に示すような方法で，数字の確認を行います．入力した数字を'0'および'9'とcmp命令で比較し，'0'～'9'までの数字であることを確認します．この範囲にない場合は，入力ループから抜けて，エラー処理errへジャンプします．

```
    cmp al,'0'              数字0と比較
    jl  err                 0より小さいなら，数字ではない
    cmp al,'9'              数字9と比較
    ja  err                 9より大なら，数字ではない
            .....
err:                        エラーの場合，ここへジャンプ
```

リスト11．11●10進入力を2進コードに変換，表示：d10to2.asm

```
 1: ;
 2: ;10進入力を2進コードに変換，表示
 3: ;
 4: .model  small
 5: ;
 6: ;-----------------------------------------------
 7: .data                            ;データセグメント
 8: ;
 9: dgtmax  equ 4                    ;最大桁数
10: digit   dw  ?                    ;入力桁数
11: bin     dw  ?                    ;2進コード
12: n10     dw  10                   ;数値10
13: bcd     db  dgtmax dup(?)        ;10進数を各桁ごとに保存．最大桁数分用意
14: ;-----------------------------------------------
15: .stack 100h                      ;スタックセグメント領域100h
16: ;-----------------------------------------------
17: .code                            ;コードセグメント
18:   extrn keyin:near,disp:near,crlf:near,exit:near       ;外部関数
19: ;-----------------------------
20: cvrt2   proc far                 ;main手続き
21: ;-------------------------
22:         mov ax,@data             ;データセグメントのアドレスをaxへ
23:         mov ds,ax                ;dsレジスタへアドレスを設定
24: ;------------------------
25: ;10進数字入力
26:         mov dl,'?'               ;入力促進表示?
27:         call disp                ;?を表示
28: ;
29:         mov cx,dgtmax            ;カウンタCX←最大桁数
30:         lea si,bcd               ;BCD保存アドレス
31: numin:  call keyin               ;1数字入力
```

```
32:         sub al,'0'          ;数値に変換
33:         mov [si],al         ;数値を桁ごとにBCDへ保存
34:         inc si              ;BCD保存アドレスを1増
35:         loop  numin         ;CX-1  最大桁数を超えないならループ
36:         call crlf           ;改行
37: ;-------------------------
38: ;入力桁数を求める
39:         mov digit,dgtmax    ;最大桁数をdigitへ
40:         sub digit,cx        ;最大桁数-CX→入力桁数
41: ;-------------------------
42: ;BCDを純2進数に変換. 上位BCD×10+下位BCDを繰り返す
43:         mov cx,digit        ;カウンタCX←入力桁数
44:         lea si,bcd          ;BCD保存アドレス
45:         xor ax,ax           ;2進数を構成するAXをクリア
46:         xor bh,bh           ;BHをクリア
47: lp:     mul n10             ;上位BCD×10
48:         mov bl,[si]         ;BCDををBLへ
49:         add ax,bx           ;AXに下位BCDを加算
50:         inc si              ;BCDの次の桁へ
51:         loop lp             ;CX-1  全桁処理済みでないならループ
52: ;
53:         mov bin,ax          ;変換した2進数を保存
54: ;-------------------------
55: ;1wordの2進数を表示
56:         mov dl,'>'          ;変換表示のため
57:         call disp           ;>を表示
58: ;
59:         mov ax,bin          ;表示する2進コード word
60:         xchg ah,al          ;ALで処理するため，上位桁をALへ
61:         call dispbin        ;上位桁を2進表示
62: ;
63:         mov ax,bin          ;表示する2進コード  AH上位桁  AL下位桁
64:         call dispbin        ;下位桁を2進表示
65: ;
66:         mov dl,'B'          ;2進数の最後のB
67:         call disp           ;Bを表示
68: ;-------------------------
69:         call exit           ;プロセス終了し，MS-DOSへ復帰
70: cvrt2   endp
71: ;-------------------------
72: dispbin proc near           ;1バイトの2進コードを10進数字で表示.2進数はalで受ける
73:         mov cx,8            ;カウンタを8に設定
74:         mov dh,al           ;データをdhへ転送
75: ;
```

```
76: dsp2:   xor dl,dl           ;dlをクリア
77:         rol dx,1            ;dxを左方向へローテート.1桁分がdlへ入る
78:         add dl,'0'          ;数字に変換するため,dlの数値に文字0を加算
79:         call disp           ;2進1桁を表示
80:         loop dsp2           ;(cx)←(cx)-1 cxが0までdsp2からを繰り返す
81: ;
82:         ret
83: dispbin endp
84: ;--------------------------------------------------------------
85:         end cvrt2           ;プログラムの終了.cvrt2番地から実行
```

実行例

```
    C:¥prg>d10to2
    ?9999
    >0010011100001111B
```

11.11 レジスタ表示

セグメントレジスタの内容をディスプレイに16進表示する.

コンピュータを利用するとき，セグメントレジスタがどのような値をとっているかを知るため，ここではセグメントレジスタの内容を表示する例を取り上げます．マクロ定義を用いてプログラムを簡単化します.

セグメントレジスタの内容を16進表示するプログラムを**リスト11．12**に示します．2進コードを16進表示するプログラムの応用です.

行6:～17:までがデータの定義で，htableは4ビットの2進数をxchg命令を用いて16進数字に変換するためのテーブルで，16進数字'0123456789ABCDEF'を初期値として与えます．hexは1バイトの2進数を上下4バイトずつにするときに用いるため，数値16を与えます．hexhとhexlは，1バイトのデータを4ビットに分解し，上下4ビットを保存するための変数で，?によって初期化されません．dsnm，esnm，ssnm，spnm，csnmはレジスタ名を表示するための文字列で，MS-DOSの文字列表示ファンクションを用いるた

め，文字列の最後の印$を付けてあります．

行19:はスタックセグメントの宣言で，領域として100Hバイトをとっています．

行21:〜36:はレジスタの内容を表示するプログラムdspregをマクロ定義しているもので，レジスタとレジスタ名が引数となっています．最初にレジスタ名のアドレスをDXレジスタに設定し，基本サブルーチンdispsで文字列を表示します．その後レジスタの上位バイトと下位バイトに分けて分解と表示を行います．バイトデータの分解はサブルーチンsepbyte，分解されたバイトデータの表示はdspbyteを呼び出すことで，行います．

行38:〜45:では，外部関数の宣言，データセグメントとエクストラセグメントのアドレスを，DSとESレジスタへ設定しています．ここで，データセグメントとエクストラセグメントは同じ領域にとることにします．

行48:〜52:では，マクロ命令によって，DS，SS，ES，CSレジスタの内容と，参考のためにSPの内容を表示します．

行58:〜64:はサブルーチンsepbyteで，1バイトの2進コードを4ビットに分解し，上位バイトを変数hexhへ，下位バイトをhexlへ保存します．簡単にするために，変数hexに保存した数値16で除算を行って，この分解をしています．

行67:〜73:はサブルーチンdspbyteで，hexhとhexlに保存された2進コードを16進表示します．表示はサブルーチンdsphexを呼びだして表示を行っています．

行76:〜83:は4ビットの2進コードを16進数字で表示するサブルーチンdsphexです．**リスト11．10**の2進コードを16進表示の表示部分をサブルーチンにしたものです．2進コードはALレジスタで受け取ります．BXレジスタに16進数字のテーブルhtableのアドレスを設定し，xlat命令によって，ALの数値に対応する16進数字に変換します．変換した数字を基本サブルーチンdispによって，表示します．

実行例を示してありますが，DSとESの値は一致しています．またスタックポインタSPはボトムの100H番地を指しています．

リスト11．12●セグメントレジスタの表示：seg.asm

```
 1: ;
 2: ;セグメントレジスタの表示
 3: ;
 4: .model  small
 5: ;------------------------------------------------
 6: .data                        ;データセグメント
 7: ;
 8: htable  db  '0123456789ABCDEF'
 9: hex     db  16               ;数値16
10: hexh    db  ?                ;16進数の上位4bit
11: hexl    db  ?                ;16進数の下位4bit
12: ;
13: dsnm    db  'DS:$'           ;データセグメント名
14: esnm    db  'ES:$'           ;エクストラセグメント名
15: ssnm    db  'SS:$'           ;スタックセグメント名
16: spnm    db  'SP:$'           ;スタックポインタ名
17: csnm    db  'CS:$'           ;コードセグメント名
18: ;------------------------------------------------
19: .stack 100H                  ;スタックセグメント領域100H
20: ;------------------------------------------------
21: ;マクロ定義                   ;引数　レジスタ，レジスタ名
22: dspreg  macro reg,regnm      ;レジスタの内容を表示
23:         lea dx,regnm         ;レジスタ名
24:         call disps           ;レジスタ名の表示
25: ;
26:         mov ax,reg           ;上位1byteを表示
27:         xchg ah,al           ;上位アドレス表示のため，上下byteを交換
28:         call sepbyte         ;上位1byteを分解
29:         call dspbyte         ;上位1byteを表示
30: ;
31:         mov ax,reg           ;下位1byteを表示
32:         call sepbyte         ;下位1byteを分解
33:         call dspbyte         ;下位1byteを表示
34: ;
35:         call crlf            ;改行
36:         endm
37: ;------------------------------------------------
38: .code                        ;コードセグメント
39:   extrn disp:near,crlf:near,disps:near,exit:near       ;外部関数
40: ;------------------------------
41: dispseg proc far             ;main手続き
42: ;--------------------------
43:         mov ax,@data         ;データセグメントのアドレスをaxへ
```

```
44:         mov ds,ax          ;DSへアドレス設定
45:         mov es,ax          ;ESへアドレス設定
46: ;-------------------------
47: ;表示
48:         dspreg ds,dsnm     ;DSの表示
49:         dspreg ss,ssnm     ;SSの表示
50:         dspreg sp,spnm     ;SPの表示
51:         dspreg es,esnm     ;ESの表示
52:         dspreg cs,csnm     ;CSの表示
53: ;-------------------------
54:         call exit          ;プロセス終了し，MS-DOSへ復帰
55: dispseg endp               ;手続きmainの終了
56: ;-------------------------
57: ;1byteの2進コードを4bitに分解
58: sepbyte proc near          ;2進数はALで受ける
59:         xor ah,ah          ;ahをクリア
60:         div hex            ;(ah:al)/16 余りは16進下位桁
61:         mov hexl,ah        ;余りを下位桁として保存
62:         mov hexh,al        ;商を上位桁として保存
63:         ret
64: sepbyte endp
65: ;-------------------------
66: ;1byteの2進コードを表示
67: dspbyte proc near          ;1byte分のデータ表示
68:         mov al,hexh        ;上位4bit
69:         call dsphex        ;16進数字で表示
70:         mov al,hexl        ;下位4bit
71:         call dsphex        ;16進数字で表示
72:         ret
73: dspbyte endp
74: ;-------------------------
75: ;4bitの2進コードを16進数字で表示.
76: dsphex  proc near          ;2進数はalで受ける
77:         mov dh,al          ;データをdhへ転送
78:         lea  bx,htable     ;テーブルの先頭番地をbxにセット
79:         xlat               ;bxが指すテーブルの(al)番目の項目をalへ
80:         mov  dl,al         ;表示のためデータをdlへ
81:         call disp          ;1文字表示
82:         ret
83: dsphex  endp
84: ;-------------------------------------------------
85:         end dispseg        ;プログラムの終了.dispsegから実行
```

実行例

```
C:¥prg>seg
DS:1BFC
SS:1C00
SP:0100
ES:1BFC
CS:1BEE
```

11.12 メモリダンプ

メモリデータをディスプレイに16進表示する．

コンピュータを利用するとき，メモリ上のデータがどのようになっているか知らなければならない場合があります．ここではメモリデータをダンプする例を取り上げます．

メモリ上の16バイトの2進コードを16進表示する例を，**リスト11．13**に示します．行6:～11:までがデータの定義になります．htableは4ビットの2進数をxchg命令を用いて16進数字に変換するためのテーブルで，16進数字'0123456789ABCDEF'を初期値として与えます．hexは1バイトの2進数を上下4バイトずつにするときに用いるため，数値16を与えます．hexhとhexlは，1バイトのデータを4ビットに分解し，上下4ビットを保存するための変数で，?によって初期化されません．

行13:～21:では，前例題と同様にスタックの設定，外部関数の宣言とデータセグメントのアドレスをDSレジスタへ設定しています．

行23:～24:では，表示するメモリアドレスと，カウンタとして用いるCXレジスタを設定しています．この例では，このプログラム自身のコード(機械語)を先頭から16バイト分表示するため，CXを16で初期化します．

行25:～47:まではループになっており，CXに設定した16回繰り返して，

アドレス:コード

の形で表示します．

行27:～37:ではアドレスの表示を行います．アドレスは2バイトあるため，上下1バイトずつ表示します．まずAXレジスタにアドレスを保持しているSIレジスタの値を代入し，上位バイトを表示するためxchg命令を用いてALレジスタへ上位バイトを設定します．これを1バイトを上下4ビットに分解するsepbyteと，上下4ビットを表示するdspbyteを呼んで，分解表示を行います．アドレスの最後に記号:を表示するため基本サブルーチンdispを用います．

コードの表示は行41:～44:で行っています．行41:の命令

```
mov al,cs:[si]
```

では，オーバライドプリフィックスCS:を用いて，ソースオペランドのデータがコードセグメントにあると指示します．このCS:がないと，SIで間接的に示されるのはデータセグメントのアドレスということになります．オーバライドプリフィックスはCS:，DS:，SS:，ES:を指定することが可能で，これにより強制的にアドレスの指すセグメントを変更することができます．

コードの表示はサブルーチンsepbyteとdspbyteを用いて行います．コードの表示後は基本サブルーチンcrlfを呼んで，ディスプレイを改行します．

行46:～47:では，次の表示アドレスを示すためSIレジスタをインクリメントして，loop命令で16バイト分の表示を繰り返します．

sepbyte，dspbyte，dsphexは前例と同じサブルーチンですので，説明を省略します．

実行例に示すように，コードセグメントの0～F番地までのメモリ状態が表示されています．これは，このプログラムをアセンブルしたコードの0～F番地のコードと一致しています．

リスト11．13●メモリダンプ：dump.asm

```
1:  ;
2:  ;メモリダンプ
3:  ;
4:  .model  small
5:  ;--------------------------------------------
6:  .data                         ;データセグメント
```

```
7: ;
8: table  db  '0123456789ABCDEF'
9: hex    db  16               ;数値16
10: hexh  db  ?                ;16進数の上位4bit
11: hexl  db  ?                ;        下位4bit
12: ;--------------------------------------------
13: .stack    100h             ;スタックセグメント領域100h
14: ;--------------------------------------------
15: .code                      ;コードセグメント
16: extrn disp:near,crlf:near,exit:near   ;外部関数
17: ;--------------------------
18: dump   proc far            ;main手続き
19: ;-----------------------
20:        mov ax,@data        ;データセグメントのアドレスをaxへ
21:        mov ds,ax           ;dsレジスタへアドレスを設定
22: ;-----------------------
23:        lea si,dump         ;表示するメモリアドレス
24:        mov cx,16           ;16byte分のカウンタ
25: lp:                        ; address:code の形式で表示
26: ;アドレスの表示
27:        mov ax,si           ;アドレスの上位1byteを表示
28:        xchg ah,al          ;上位アドレス表示のため, alに上位byte
29:        call sepbyte        ;上位1byteを分解
30:        call dspbyte        ;上位1byteを表示
31: ;
32:        mov ax,si           ;アドレスの下位1byteを表示
33:        call sepbyte        ;下位1byteを分解
34:        call dspbyte        ;下位1byteを表示
35:        ;
36:        mov dl,':'          ;文字:
37:        call disp           ;アドレスの後に:を表示
38: ;
39: ;-----------------------
40: ;コードの表示
41:        mov al,cs:[si]      ;コードをalへ転送
42:        call sepbyte        ;1byteを分解
43:        call dspbyte        ;下位1byteを表示
44:        call crlf           ;改行
45: ;
46:        inc si              ;表示アドレスを1増
47:        loop lp             ;16byte表示していないならループ
48: ;
49:        call exit           ;プロセス終了し, MS-DOSへ復帰
50: dump   endp                ;手続きmainの終了
```

```
51: ;--------------------------
52: ;1byteの2進コードを4bitに分解
53: sepbyte proc near         ;2進数はalで受ける
54:         xor ah,ah         ;ahをクリア
55:         div hex           ;(ah:al)/16 余りは16進下位桁
56:         mov hexl,ah       ;余りを下位桁として保存
57:         mov hexh,al       ;商を上位桁として保存
58:         ret
59: sepbyte endp
60: ;--------------------------
61: ;1byteの2進コードを表示
62: dspbyte proc near         ;1byte分のデータ表示
63:         mov al,hexh       ;上位4bit
64:         call dsphex       ;16進数字で表示
65:         mov al,hexl       ;下位4bit
66:         call dsphex       ;16進数字で表示
67:         ret
68: dspbyte endp
69: ;--------------------------
70: ;4bitの2進コードを16進数字で表示.
71: dsphex  proc near         ;2進数はalで受ける
72:         mov dh,al         ;データをdhへ転送
73:         lea  bx,table     ;テーブルの先頭番地をbxにセット
74:         xlat              ;bxが指すテーブルの(al)番目の項目をalへ
75:         mov  dl,al        ;表示のためデータをdlへ
76:         call disp         ;1文字表示
77:         ret
78: dsphex  endp
79: ;----------------------------------------------
80: end dump                  ;プログラムの終了.dump番地から実行
```

実行例

```
C:¥prg>dump
0000:B8
0001:F6
0002:1B
0003:8E
0004:D8
0005:8D
0006:36
0007:00
0008:00
0009:B9
```

```
000A:10
000B:00
000C:8B
000D:C6
000E:86
000F:E0
```

11.13 アンパック型10進数の加算

キーボードから2つの10進数を入力し，アンパック型10進数として加算表示する．10進数は5桁以内とする．

キーボードから5桁以下の10進数を2つ入力し，各桁をアンパック型10進数に変換した後，加算し結果を表示します．プログラムをわかりやすくするためマクロ定義とサブルーチンを用いますが，これをインクルッドファイルに保存します．インクルッドファイルはメインプログラムと別にアセンブルするのではなく，メインプログラムのアセンブル時にインクルッドファイルが読み込まれて，1つのプログラムとしてアセンブルされます．

プログラム例を**リスト11．14**に示します．行6:～13:までがデータの定義で，変数dgtmaxは10進数の最大入力桁数，dgtxとdgtyは実際に入力された被加数xと加数yの桁数を保持します．bufxとbufyは入力時のバッファとして用いる配列で，配列xとyはこれをBCD各桁に正規化したものです．xは被加数ですが加算結果もここに保存します．

行23:～25:ではデータセグメントレジスタDSとエクストラセグメントレジスタESのアドレスを設定します．このプログラムではストリング命令を用いるため，DSとESを同領域にとる必要があります．

行27:～34:では，マクロ命令を用いて領域のクリア，10進数の入力を行います．まずマクロclearを用いてxとyの配列を0にクリアします．パラメータxとyは配列名，dgtmax＋1は配列の大きさを示します．配列の大きさは入力桁数の最大dgtmaxより1だけ多くとってあります．これは，入力した最上位桁の加

算で桁上げが生じた場合に備えるものです．

次にマクロindecを用いてバッファbufxに被加数，bufyに加数を入力します．被加数の入力のため'?'を，加数入力時に'＋'を表示するため，これをパラメータとして渡しています．また入力桁数を保存するために，桁数を保持する変数dgtxとdgtyをパラメータとして渡します．この入力ではバッファbufxとbufyの最上位桁(most significant digit MSD)から入力されます．2つの入力の桁数が異なる場合は，単純にバッファの対応する桁同士を加算することができません．そこでマクロmovbcdを用いて，xとyの最下位桁(least significant digit LSD)が同桁になるようにbufxとbufyから転送します．

実際のアンパック加算は行36:～46:で行います．加算にadc命令を用いるため，最初にキャリフラグCFをクリアし，ストリング命令でのアドレスの増減を，減少方向(下位桁から処理)にとるためディレクションフラグDFをセットします．

lea命令を用いてレジスタSIに被加数xの最下位桁アドレスを設定し，DIに加数yの最下位桁を設定します．加算桁数のカウンタとして用いるレジスタCXには，MSDの加算で桁上げが生ずることを考慮して，最大入力桁数dgtmax＋1を与えます．

行41:～46:のループでBCD各桁の加算を下位桁から行います．ストリング命令lodsbで被加数xの最下位桁をALへロードし，adc命令でx＋y＋CFを行います．ここで，CFには前回の加算で生じた桁上げが保存されていますが，ループの第1回目ではCFはクリアされています．その後aaa命令で10進加算後の補正を行い，mov [si]＋1,alで加算結果をxの該当する桁へ保存します．ここで保存アドレスが[si]＋1となっているのは，lodsb命令が実行された後，自動的にSIの内容がディクリメント(1減)になっているためです．次に加数アドレスを保持するDIをディクリメントして次のループに備えます．loop命令で最大入力桁数＋1だけループ処理します．なおloop命令ではCXの値が自動的にディクリメントされますが，この処理ではCFには影響を与えません．

行48:では，マクロdspdecを参照して加算の解を表示します．このとき表示する＝と，解を保持するxをパラメータとして渡しています．

実行例に示すように，最上位桁で起こった桁上げにも対応して答えが表示されています．

リスト11．14●アンパック型10進加算：pdadd.asm

```
 1:         ;
 2:         ;  アンパック型10進加算
 3:         ;
 4:         .model small
 5:         ;--------------------------------------------
 6:         .data                   ;データセグメント
 7:         dgtmax equ 5            ;最大入力桁数
 8:         dgtx dw  ?              ;入力桁数x
 9:         dgty dw  ?              ;入力桁数y
10: bufx db dgtmax dup(?)       ;入力バッファx  配列
11: bufy db dgtmax dup(?)       ;入力バッファy  配列
12: x db dgtmax+1 dup(?)        ;被加数BCDx各桁 配列
13: y db gtmax+1 dup(?)         ;  加数BCDy各桁 配列
14: ;--------------------------------------------
15: .stack 100H                 ;スタックセグメント領域100H
16: ;--------------------------------------------
17: .code                       ;コードセグメント
18:   extrn keyin:near,disp:near,crlf:near,exit:near ;外部関数
19: include dec.inc             ;インクルッドファイル
20: ;------------------------------
21: decadd proc far             ;main手続き
22: ;-------------------------
23:         mov ax,@data        ;データセグメントのアドレスをaxへ
24:         mov ds,ax           ;DSレジスタへアドレスを設定
25:         mov es,ax           ;ESレジスタへアドレスを設定
26: ;------------------------
27:         clear x,dgtmax+1    ;被加数(BCD)をクリア(マクロ)
28:         clear y,dgtmax+1    ;  加数(BCD)をクリア(マクロ)
29: ;
30:         indec '?',bufx,dgtx    ;被加数入力(マクロ)
31:         indec '+',bufy,dgty    ;  加数入力(マクロ)
32: ;
33:         movbcd x,bufx,dgtx ;bufxからxへ転送(マクロ)
34:         movbcd y,bufy,dgty ;bufyからyへ転送(マクロ)
35: ;------------------------
36:         clc                 ;キャリフラグをクリヤ
37:         std                 ;方向を減少に
38:         lea si,x+dgtmax     ;被加数のアドレス設定.下位桁から処理
39:         lea di,y+dgtmax     ;  加数のアドレス設定
40:         mov cx,dgtmax+1     ;加算する桁数 桁上げを考慮して1桁増
41: alp:    lodsb               ;被加数をロード，その後アドレスを1減
42:         adc al,[di]         ;加算  x+y+桁上げCF
43:         aaa                 ;アンパック10進補正.桁上げをCFへセット
```

```
44:         mov [si]+1,al      ;加算結果を保存
45:         dec di             ;加数アドレスを1減
46:         loop alp           ;最大入力桁数+1だけ繰返し.フラグには影響しない
47: ;-------------------------
48:         dspdec '=',x       ;解を表示(マクロ参照)
49: ;-------------------------
50:   call exit                ;プロセス終了し，MS-DOSへ復帰
51: decadd endp
52: ;-----------------------------------------------------------
53:   end decadd               ;プログラムの終了.decadd番地から実行
```

実行例

```
C:¥prg>pdadd
?12345
+98765
=111110
```

マクロとサブルーチンを定義したインクルッドファイルを，**リスト11．15**に示します．行8:～14:がメモリ上の配列をクリアするマクロclearで，パラメータとして，配列名とそのバイト数をとります．配列のアドレスをレジスタDIへ，カウンタとして用いるCXへバイト数を設定して，ストリング命令stosbで領域をクリアします．このときプリフィックスrepがつくと，stosb命令はCXに保存された値だけ繰り返し実行されます．

行17:～24は配列の要素を転送するマクロmovbcdで，パラメータとして転送先，転送元の配列，バイト数がとられます．転送するバイト数をCXへ設定して，SIへ転送元の実際上の最下位桁アドレスを設定します．DIへは転送先の最下位桁アドレスを設定して，repプリフィックスを付加したストリング命令movsbで，入力した桁数だけ転送を繰り返します．

行27:～34:は10進数の入力を行うマクロindecで，パラメータとして表示記号，バッファ，入力桁数を保存する変数をとります．最初に入力促進記号を表示して，入力バッファアドレスをSIに，入力桁数を保持するアドレスをDIに設定して，実際にキー入力するサブルーチンkindecを呼びます．

行37:～43:は10進で解を表示するマクロdspdecで，パラメータとして表示記号と解を持つ配列をとります．最初に記号を表示して，解のある配列アドレ

スをSIに保持して，実際に10進表示をするサブルーチンdispbcdを呼びます．

行45:からはサブルーチンの定義で，行48:〜63:は10進数字をキーボードから入力するサブルーチンkindecで，SIに入力バッファアドレスを保持してあることが必要です．最初に基本サブルーチンkeyinを呼んで1数字を入力し，数字0〜9であることを判断します．入力数字がこの範囲にあるならば，この数字から'0'を引いて4ビットBCDに変換し，指定されたバッファに桁ごとに保存します．これを最大入力桁数dgtmax回か，数字でない文字が入力されるまで繰り返します．最後に実際に入力された桁数を指定された変数(DIで間接アドレス)へ保存します．

行66:〜74:はBCDコードを10進表示するサブルーチンdispbcdです．SIに表示するBCDアドレスが保持されていることが必要です．表示する桁数は，入力の最大入力桁数＋1でMSDでの加算で桁上げが生じた場合にも対応できるようにしたものです．ロードしたBCD各桁に'0'を加算して数字に変換します．これをloop命令でdgtmax＋1回繰り返して，全桁を表示します．

リスト11．15●アンパック型10進演算のマクロとサブルーチン（dec.inc）

```
 1 ;
 2: ;  インクルッドファイル  アンパック型10進演算
 3: ;
 4: ;------------------------------------------
 5: ;マクロ定義
 6: ;--------------------------
 7: ;メモリ（配列）をクリア
 8: clear  macro  mem,dgts    ;パラメータ：配列名,バイト数
 9:        cld                ;方向を増加に
10:        lea di,mem         ;配列のアドレス
11:        xor al,al          ;ALをクリア
12:        mov cx,dgts        ;バイト数
13:   rep  stosb              ;繰り返し配列の全要素を0
14: endm
15: ;--------------------------
16: ;配列の要素を転送
17: movbcd macro xy,bufxy,dgtxy  ;パラメータ 転送先,転送元,バイト数
18:        std                ;方向を減少に
19:        mov cx,dgtxy       ;バイト数をカウンタにセット
20:        lea si,bufxy-1     ;転送元のアドレスを設定
21:        add si,dgtxy       ;入力された桁数を加算．実際の最下位桁(LSD)から処理
```

```
22:         lea di,xy+dgtmax   ;転送先のLSDアドレスを設定
23:   rep   movsb              ;入力桁数だけ転送を繰り返す
24:         endm
25: ;--------------------------
26: ;10進数の入力
27: indec   macro  sym,bufxy,dgtxy ;パラメータ：表示記号,バッファ,桁数
28:         mov dl,sym         ;入力促進記号
29:         call disp          ;記号を表示
30:         lea si,bufxy       ;入力保存アドレス
31:         lea di,dgtxy       ;入力桁数
32:         call kindec        ;10進キー入力
33:         call crlf          ;ディスプレイ改行
34:         endm
35: ;--------------------------
36: ;解の表示
37: dspdec macro  sym,ans      ;パラメータ：表示記号,解
38:         mov dl,sym         ;表示記号
39:         call disp          ;記号を表示
40:         lea si,ans         ;解のアドレス
41:         call dispbcd       ;解BCDの表示
42:         call crlf          ;改行
43:         endm
44: ;----------------------------------------------
45: ;サブルーチン
46: ;--------------------------
47: ;10進数字キー入力
48: kindec  proc near          ;SIに入力バッファアドレス
49:         mov cx,dgtmax      ;カウンタCX←最大入力桁数
50: numin:  call keyin         ;1数字入力
51:         cmp al,'0'         ;数字0と比較
52:         jl brk             ;0より小さいなら，数字ではない
53:         cmp al,'9'         ;数字9と比較
54:         ja brk             ;9より大なら，数字ではない
55:         sub al,'0'         ;数値BCDに変換
56:         mov [si],al        ;数値を桁ごとにバッファへ保存
57:         inc si             ;バッファアドレスを1増
58:         loop  numin        ;CX-1  最大入力桁数を超えないならループ
59: brk:                       ;入力桁数を求める
60:         mov [di],dgtmax    ;最大入力桁数をdgtmaxへ
61:         sub [di],cx        ;最大入力桁数-CX=入力桁数
62:         ret
63: kindec  endp
64: ;--------------------------
65: ;10進数の表示
```

```
66: dispbcdproc near           ;BCDを10進数字で表示.SIにBCDアドレス
67:         mov cx,dgtmax+1    ;カウンタに最大入力桁数+1を設定
68: dslp:    mov dl,[si]       ;データをDLへ転送
69:         add dl,'0'         ;数字に変換するため，DLに文字0を加算
70:         call disp          ;1桁を表示
71:         inc si             ;BCDアドレスを1増
72:         loop dslp          ;最大入力桁数だけ繰り返す
73:         ret
74: dispbcd endp
75: ;-----------------------------------------------------------
```

11.14 アンパック型10進数の減算

キーボードから2つの10進数を入力し，アンパック型10進数として減算し，表示する．10進数は5桁以内とする．

キーボードから5桁以下の10進数を2つ入力し，各桁をアンパック型10進数に変換した後，減算し結果を表示します．前例題と同じマクロ定義とサブルーチンをインクルッドファイルから読み込みます．

アンパック型10進減算のプログラム例を**リスト11．16**に示します．行6:～13:までがデータの定義で，変数dgtmaxは10進数の最大入力桁数，dgtxとdgtyは実際に入力された被減数xと減数yの桁数を保持します．bufxとbufyは入力時のバッファとして用いる配列で，配列xとyはこれをBCD各桁に正規化したものです．xは被減数ですが減算結果もここに保存します．

行23:～25:ではデータセグメントレジスタDSとエクストラセグメントレジスタESのアドレスを設定します．このプログラムもストリング命令を用いるため，DSとESを同領域にとる必要があります．

行27:～34:では，マクロ命令を用いて領域のクリア，10進数の入力を行います．まずマクロclearを用いてxとyの配列を0にクリアします．マクロへのパラメータxとyは配列名，dgtmax＋1は配列の大きさを示します．配列の大きさは入力桁数の最大dgtmaxより1だけ多くとってあります．これは，入力した

最上位桁の減算で借りが生じた場合に備えるものです．

次にマクロindecを用いてバッファbufxに被減数，bufyに減数を入力します．被減数の入力のため'?'を，減数入力時に'－'を表示するため，これをパラメータとして渡しています．また入力桁数を保存するために，桁数を保持する変数dgtxとdgtyをマクロのパラメータとして渡します．この入力ではバッファbufxとbufyの最上位桁(MSD)から入力されます．2つの入力の桁数が異なる場合は，単純に対応する桁同士を減算することができません．そこでマクロmovbcdを用いて，xとyの最下位桁(LSD)が同桁になるようにbufxとbufyから転送します．

実際のアンパック減算は行36:～46:で行います．減算にsbb命令を用いるため，最初にキャリフラグCFをクリアし，ストリング命令でのアドレスの増減を，減少方向(下位桁から処理)にとるためディレクションフラグDFをセットします．

lea命令を用いてレジスタSIに被減数xの最下位桁アドレスを設定し，DIに減数yの最下位桁を設定します．減算桁数のカウンタとして用いるレジスタCXには，MSDの減算で借りが生ずることを考慮して，最大入力桁数dgtmax＋1を与えます．

行41:～46:のループでBCD各桁の減算を下位桁から行います．ストリング命令lodsbで被減数xの最下位桁をALへロードし，sbb命令でx－y－CFを行います．ここで，CFには前回の減算で生じた借りが保存されていますが，ループの第1回目ではCFはクリアされています．その後aas命令でBCD減算後の補正を行い，mov [si]＋1,alで減算結果をxの該当する桁へ保存します．ここで保存アドレスが[si]＋1となっているのは，lodsb命令が実行された後，自動的にSIの内容がディクリメント(1減)されるためです．次に減数アドレスを保持するDIをディクリメントして次のループに備えます．loop命令で最大入力桁数＋1だけループ処理しますが，このときCXの値が自動的にディクリメントされるにも関わらず，この処理ではCFには影響を与えません．

行48:では，マクロdspdecを参照して減算結果を表示します．このとき表示する＝と解を保持するxをパラメータとしてマクロに渡しています．

実行例では5桁－4桁の減算を行っていますが，結果が負になった場合の処理はこのプログラムでは行っていません．

リスト11．16●アンパック型10進減算：pdsub.asm

```
 1: ;
 2: ;  アンパック型10進減算
 3: ;
 4: .model  small
 5: ;--------------------------------------------
 6: .data                       ;データセグメント
 7: dgtmax  equ 5               ;最大入力桁数
 8: dgtx    dw  ?               ;入力桁数x
 9: dgty    dw  ?               ;入力桁数y
10: bufx    db  dgtmax dup(?)   ;入力バッファx  配列
11: bufy    db  dgtmax dup(?)   ;入力バッファy  配列
12: x  db  dgtmax+1 dup(?)      ;被減数BCDx各桁 配列
13: y  db  dgtmax+1 dup(?)      ;  減数BCDy各桁 配列
14: ;--------------------------------------------
15: .stack 100H                 ;スタックセグメント領域100H
16: ;--------------------------------------------
17: .code                       ;コードセグメント
18:   extrn keyin:near,disp:near,crlf:near,exit:near ;外部関数
19:   include pr_dec.inc        ;インクルッドファイル
20: ;-------------------------
21: decsub  proc far            ;main手続き
22: ;-------------------------
23:         mov ax,@data        ;データセグメントのアドレスをaxへ
24:         mov ds,ax           ;DSレジスタへアドレスを設定
25:         mov es,ax           ;ESレジスタへアドレスを設定
26: ;-------------------------
27:         clear x,dgtmax+1    ;被減数(BCD)をクリア(マクロ参照)
28:         clear y,dgtmax+1    ;  減数(BCD)をクリア(マクロ参照)
29: ;
30:         indec '?',bufx,dgtx    ;被減数入力(マクロ参照)
31:         indec '-',bufy,dgty    ;  減数入力(マクロ参照)
32: ;
33:         movbcd x,bufx,dgtx ;bufxからxへ転送(マクロ参照)
34:         movbcd y,bufy,dgty ;bufyからyへ転送(マクロ参照)
35: ;-------------------------
36:         clc                 ;キャリフラグをクリヤ
37:         std                 ;方向を減少に
38:         lea si,x+dgtmax     ;被減数のアドレス設定.下位桁から処理
39:         lea di,y+dgtmax     ;  減数のアドレス設定
40:         mov cx,dgtmax+1     ;減算する桁数 桁上げを考慮して1桁増
41: slp:    lodsb               ;被減数をロード，その後アドレスを1減
42:         sbb al,[di]         ;加算  x+y+桁上げCF
43:         aas                 ;アンパック10進補正.桁上げをCFへセット
```

```
44:         mov [si]+1,al       ;加算結果を保存
45:         dec di              ;減数アドレスを1減
46:         loop slp            ;桁数+1だけ繰返し.フラグには影響しない
47: ;-------------------------
48:         dspdec '=',x        ;解を表示(マクロ参照)
49: ;
50:         call exit           ;プロセス終了し,MS-DOSへ復帰
51: decsub  endp
52: ;----------------------------------------------------------------
53:         end decsub          ;プログラムの終了.decsub番地から実行
```

実行例

```
C:¥prg>pdsub
?34567
-9876
=024691
```

11.15 アンパック型10進数の乗算

キーボードから2つの10進数を入力し，アンパック型10進数として乗算し，表示する．被乗数は5桁以内，乗数は1桁とする．

キーボードから5桁以内の被乗数と1桁の乗数を10進数として入力し，各桁をアンパック型10進数に変換した後，乗算し結果を表示します．簡単にするために乗数は10進1桁としてあります．また前例題と同じく，マクロ定義とサブルーチンをインクルッドファイルから読み込みます．

アンパック型10進乗算のプログラム例を**リスト11．17**に示します．行6:～13:までがデータの定義で，変数dgtmaxは10進数の最大入力桁数，dgtxとdgtyは実際に入力された被乗数xと乗数yの桁数を保持します．bufxとbufyは入力時のバッファとして用いる配列で，配列xとyはこれをBCD各桁に正規化したものです．xは被乗数ですが乗算結果もここに保存します．

行24:～26:ではデータセグメントレジスタDSとエクストラセグメントレジスタESのアドレス設定で，このプログラムでもストリング命令を用いるため，DS

とESを同領域にとる必要があります．

行28:～35:では，マクロclearによって領域のクリア，indecによって10進数をバッファへ入力，movbcdによってバッファから被乗数xと乗数yへ転送して正規化します．なお，ここでもxは入力した最上位桁で桁上げが生じた場合に備え，最大入力桁数dgtmaxより1だけ多くとってあります．この例でyは1桁であるにも関わらず，桁数を拡張することを考慮して，xと同じ桁数を用意してあります．

実際のアンパック型乗算は行37:～48:で行います．ストリング命令でのアドレスの増減を，減少方向(下位桁から処理)にとるためディレクションフラグDFをセットします．

lea命令を用いてレジスタSIに，被乗数xの最下位桁アドレスを設定し，DIに乗数yの最下位桁アドレスを設定します．乗算桁数のカウンタとして用いるレジスタCXには，最上位桁での桁上げが生ずることを考慮して，最大入力桁数dgtmax＋1を与えます．

行41:～48:のループで，BCD各桁の乗算を下位桁から行います．ストリング命令lodsbで被乗数xの最下位桁をALへロードし，mul命令でx×yを行います．その後aam命令でBCD乗算後の補正を行います．さらに前回乗算で桁上げが生じていることを考慮して，add命令で桁上げを加算し，ここでまた10進補正命令aaaで補正します．この後，さらにmov [si]＋1,alで乗算結果をxの該当する桁へ保存します．ここで保存アドレスが[si]＋1となっているのは，lodsb命令が実行された後，自動的にSIの内容がディクリメント(1減)されるためです．loop命令で最大入力桁数＋1だけループ処理しますが，このときCXの値が自動的にディクリメントされるにも関わらず，この処理ではCFには影響を与えません．

行50:では，マクロdspdecを参照して乗算結果を表示します．

実行例では5桁×1桁の乗算を行っていますが，結果が6桁になった場合も，乗算結果は対応しています．

リスト11．17●アンパック型10進乗算：pdmul.asm

```
 1: ;
 2: ;  アンパック型10進乗算
 3: ;
 4: .model  small
 5: ;-----------------------------------------------
 6: .data                        ;データセグメント
 7: dgtmax  equ 5                ;最大入力桁数
 8: dgtx    dw  ?                ;入力桁数x
 9: dgty    dw  ?                ;入力桁数y
10: bufx    db  dgtmax dup(?)    ;入力バッファx  配列
11: bufy    db  dgtmax dup(?)    ;入力バッファy  配列
12: x       db  dgtmax+1 dup(?)     ;被乗数BCDx各桁 配列
13: y       db  dgtmax+1 dup(?)     ;  乗数BCDy各桁 配列
14: carry   db ?                 ;桁上げ
15: ;-----------------------------------------------
16: .stack 100H                  ;スタックセグメント領域100H
17: ;-----------------------------------------------
18: .code                        ;コードセグメント
19:   extrn keyin:near,disp:near,crlf:near,exit:near ;外部関数
20:   include pr_dec.inc            ;インクルッドファイル
21: ;--------------------------------
22: decmul  proc far                ;main手続き
23: ;--------------------------
24:         mov ax,@data         ;データセグメントのアドレスをaxへ
25:         mov ds,ax            ;DSレジスタへアドレスを設定
26:         mov es,ax            ;ESレジスタへアドレスを設定
27: ;-------------------------
28:         clear x,dgtmax+1     ;被乗数(BCD)をクリア(マクロ参照)
29:         clear y,dgtmax+1     ;  乗数(BCD)をクリア(マクロ参照)
30: ;
31:         indec '?',bufx,dgtx     ;被乗数入力(マクロ参照)
32:         indec '*',bufy,dgty     ;  乗数入力(マクロ参照)
33: ;
34:         movbcd x,bufx,dgtx ;bufxからxへ転送(マクロ参照)
35:         movbcd y,bufy,dgty ;bufyからyへ転送(マクロ参照)
36: ;-------------------------
37:         std                  ;方向を減少に
38:         lea si,x+dgtmax      ;被乗数のアドレス設定
39:         mov cx,dgtmax+1      ;乗算する桁数。桁上げを考慮して1桁増
40:         mov carry,0          ;桁上げをクリア
41: mlp:    lodsb                ;被乗数をロード
42:         mul y+dgtmax         ;乗算 x·y
43:         aam                  ;アンパック10進補正
```

```
44:        add al,carry      ;乗算結果と下位からの桁上げを加算
45:        aaa               ;アンパック10進補正
46:        mov [si]+1,al     ;結果をxへ保存
47:        mov carry,ah      ;上位への桁上げを保存
48:        loop mlp          ;桁数+1だけ繰返し.フラグには影響しない
49: ;------------------------
50:        dspdec '=',x      ;解を表示(マクロ参照)
51: ;
52:        call exit         ;プロセス終了し，MS-DOSへ復帰
53: decmul endp
54: ;--------------------------------------------------------
55:        end decmul        ;プログラムの終了.decmul番地から実行
```

実行例

```
C:¥prg>pdmul
?54321
*8
=434568
```

11.16 アンパック型10進数の除算

キーボードから2つの10進数を入力し，アンパック型10進数として除算し，結果を表示する．被除数は5桁以内，除数は1桁とする．

キーボードから5桁以内の被除数と1桁の除数を10進数として入力し，各桁をアンパック型10進数に変換した後，除算し結果を表示します．簡単にするために除数は10進1桁としてあります．また前例題と同じく，マクロ定義とサブルーチンをインクルッドファイルから読み込みます．

アンパック型10進除算のプログラム例を**リスト11．18**に示します．行6:～13:までがデータの定義で，変数dgtmaxは10進数の最大入力桁数，dgtxとdgtyは実際に入力された被除数xと除数yの桁数を保持します．bufxとbufyは入力時のバッファとして用いる配列で，配列xとyはこれをBCD各桁に正規化したものです．xは被除数ですが除算結果もここに保存します．

行23:〜25:ではデータセグメントレジスタDSとエクストラセグメントレジスタESのアドレス設定で，このプログラムでもストリング命令を用いるため，DSとESを同領域にとる必要があります．

行27:〜34:では，マクロclearによって領域のクリア，indecによって10進数をバッファへ入力，movbcdによってバッファから被除数xと除数yへ転送して正規化します．ここでもxは最大入力桁数dgtmaxより1だけ多くとってあります．この例でyは1桁であるにも関わらず，桁数を拡張することを考慮してxと同じ桁数を用意してあります．

実際のアンパック型除算は行36:〜44:で行います．除算で，被除数は上位桁から処理を行います．そのためストリング命令でのアドレスの増減を，増加方向(下位桁から処理)にとるため，ディレクションフラグDFをクリアします．

lea命令を用いてレジスタSIに，被除数xの最上位桁アドレスを設定します．除算桁数のカウンタとして用いるレジスタCXには，最大入力桁数dgtmax＋1を与えています．

行40:〜44:のループでBCD各桁の除算を上位桁から行います．ストリング命令lodsbで被除数xの最上位桁をALへロードし，除算前に10進補正命令aadで補正します．その後div命令でx/yを行います．この後mov [si]－1,alで除算結果をxの該当する桁へ保存します．ここで保存アドレスが[si]－1となっているのは，lodsb命令が実行された後，自動的にSIの内容がインクリメント(1増)されるためです．loop命令で最大入力桁数＋1だけループ処理しますが，このときCXの値が自動的にディクリメントされるにも関わらず，この処理ではCFには影響を与えません．

行46:で，マクロdspdecを参照して除算結果を表示します．実行例では5桁／1桁の除算を行っています．

リスト11．18●アンパック型10進除算：pddiv.asm

```
 1: ;
 2: ;  アンパック型10進除算
 3: ;
 4: .model  small
 5: ;--------------------------------------------------
 6: .data                          ;データセグメント
```

```
 7: dgtmax equ 5              ;最大入力桁数
 8: dgtx   dw  ?              ;入力桁数x
 9: dgty   dw  ?              ;入力桁数y
10: bufx   db  dgtmax dup(?)  ;入力バッファx  配列
11: bufy   db  dgtmax dup(?)  ;入力バッファy  配列
12: x      db  dgtmax+1 dup(?)    ;被除数BCDx  配列
13: y      db  dgtmax+1 dup(?)    ;  除数BCDy  配列
14: ;-----------------------------------------
15: .stack 100H               ;スタックセグメント領域100H
16: ;-----------------------------------------
17: .code                     ;コードセグメント
18:   extrn keyin:near,disp:near,crlf:near,exit:near ;外部関数
19:   include pr_dec.inc      ;インクルッドファイル
20: ;----------------------------
21: decdiv proc far           ;main手続き
22: ;------------------------
23:        mov ax,@data       ;データセグメントのアドレスをaxへ
24:        mov ds,ax          ;DSレジスタへアドレスを設定
25:        mov es,ax          ;ESレジスタへアドレスを設定
26: ;-----------------------
27:        clear x,dgtmax+1   ;被除数(BCD)をクリア(マクロ参照)
28:        clear y,dgtmax+1   ;  除数(BCD)をクリア(マクロ参照)
29: ;
30:        indec '?',bufx,dgtx    ;被除数入力(マクロ参照)
31:        indec '/',bufy,dgty    ;除数入力(マクロ参照)
32: ;
33:        movbcd x,bufx,dgtx ;bufxからxへ転送(マクロ参照)
34:        movbcd y,bufy,dgty ;bufyからyへ転送(マクロ参照)
35: ;-----------------------
36:        cld                ;方向を増加に
37:        lea si,x           ;被除数のアドレス設定
38:        mov cx,dgtmax+1    ;除算する桁数.桁上げを考慮して1桁増
39:        mov ah,0           ;上位をクリア
40: dlp:   lodsb              ;被除数をロード
41:        aad                ;アンパック10進補正
42:        div y+dgtmax       ;除算 x/y
43:        mov [si]-1,al      ;結果をxへ保存
44:        loop dlp           ;桁数+1だけ繰返し.フラグには影響しない
45: ;-----------------------
46:        dspdec '=',x       ;解を表示(マクロ参照)
47: ;
48:        call exit          ;プロセス終了し，MS-DOSへ復帰
49: decdiv endp
50: ;-----------------------------------------------------
```

```
51:          end decdiv          ;プログラムの終了.decdiv番地から実行
```

実行例

```
C:¥>pddiv
?12345
/5
=002469
```

第11章 基本プログラミング

演習問題11

1. 次の表示に関するプログラムを作成し，実行せよ．

(1) keyinと表示してから，エコーバックなしに複数文字キー入力し，そのまま表示する．入力の終了はEnter入力を検出する．

(2) 基本サブルーチンdispsと同じ働きをするプログラムをdispを用いて作成し，あらかじめ用意した文字列を表示する．表示は文字列中の$を検出し，終了する．

(3) MS-DOSファンクションにより，現在の日付と時刻を取得し表示する．

2. 次のハードウエアに関するプログラムを作成し，実行せよ．

(1) 指定したメモリアドレスの内容をダンプする．16バイト分を1行に表示すること．

(2) 全レジスタの内容をレジスタ名と共に表示する．

(3) 上記(2)で，フラグレジスタの状態をフラグ名と共に表示するよう拡張する．

3. 次のテーブルによる変換を用いたプログラムを作成し，実行せよ．

(1) ア行のカタカナをキー入力し，ローマ字に変換して表示する．
例　ア→a

(2) ローマ字の母音をキー入力し，カタカナに変換して表示する．
例　a→ア

(3) 英大文字をキー入力し，小文字に変換して表示する．

(4) 英小文字をキー入力し，大文字に変換して表示する．

(5) 上記問題(1)～(4)の入出力文字コードを16進表示する．

4. 次のコード変換プログラムを作成し，実行せよ．

(1) 2進数（8桁）をキー入力し，8進数で表示する．

(2) 2進数（8桁）をキー入力し，10進数で表示する．

(3) 2進数（8桁）をキー入力し，16進数で表示する．

(4) 10進数（5桁）をキー入力し，2進数で表示する．

(5) 10進数（5桁）をキー入力し，8進数で表示する．

(6) 10進数（5桁）をキー入力し，16進数で表示する．

(7) 16進数（4桁）をキー入力し，2進数で表示する．

(8) 16進数（4桁）をキー入力し，8進数で表示する．

(9)　16進数（4桁）をキー入力し，10進数で表示する．
(10) 漢字（複数）をキー入力し，そのコードを16進数で表示する．

5.　次のアンパック型10進計算プログラムを作成し，実行せよ．

(1)　乗数（x×yのy）が複数桁の乗算を可能にする．
(2)　除数（x／yのy）が複数桁の除算を可能にする．
(3)　電卓機能を実現する．プログラムは段階をおって記述する．
　　①1桁入力（入力形式　1＋2＝）を可能にする．
　　②最終的には8～10桁程度を入力可能にする．
　　③符号入力を可能にする．(例　－1＋2)
　　④入力エラーを検出し，これに対応する．

付録 1　8086命令一覧

命令	オペランド	動作内容	機能	O	D	I	T	S	Z	A	P	C
AAA		if ((AL) & 0FH)>9 or (AF)=1 then (AL)←(AL)+6, (AH)←(AH)+1,(AF)←1,(CF)←(AF),(AL)←(AL) & 0FH	ASCII加算後の補正	U				U	U	X	U	X
AAD		(AL)←(AH)*0AH+(AL), (AH)←0	ASCII除算前の補正	U				X	X	U	X	U
AAM		(AH)←(AL)/0AH, (AL)←(AL)%0AH	ASCII乗算後の補正	U				X	X	U	X	U
AAS		if ((AL) & 0FH)>9 or (AF)=1 then (AL)←(AL)-6, (AH)←(AH)-1,(AF)←1,(CF)←(AF),(AL)←(AL) & 0FH	ASCII減算後の補正	U				U	U	X	U	X
ADC	reg,reg	(reg) ←(reg)+(reg)+(CF)	キャリを含めた加算	X				X	X	X	X	X
	reg,mem	(reg) ←(reg)+(mem)+(CF)										
	mem,reg	(mem) ←(mem)+(reg)+(CF)										
	reg,imm	(reg) ←(reg)+imm+(CF)										
	mem,imm	(mem) ←(mem)+imm+(CF)										
	acc,imm	if W=0 (AL)←(AL)+imm+(CF) else W=1 (AX)←(AX)+imm+(CF)										
ADD	reg,reg	(reg) ←(reg)+(reg)	加算	X				X	X	X	X	X
	reg,mem	(reg) ←(reg)+(mem)										
	mem,reg	(mem) ←(mem)+(reg)										
	reg,imm	(reg) ←(reg)+imm										
	mem,imm	(mem) ←(mem)+imm										
	acc,imm	if W=0 (AL)←(AL)+imm else W=1 (AX)←(AX)+imm										
AND	reg,reg	(reg) ←(reg) ∧ (reg)	論理積	0				X	X	U	X	0
	reg,mem	(reg) ←(reg) ∧ (mem)										
	mem,reg	(mem) ←(mem) ∧ (reg)										
	reg,imm	(reg) ←(reg) ∧ imm										
	mem,imm	(mem) ←(mem) ∧ imm										
	acc,imm	if W=0 (AL)←(AL) ∧ imm else W=1 (AX)←(AX) ∧ imm										
CALL	near-label	(SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(IP)+disp	手続き呼び出し									
	regptr16	(SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(IP)+(regptr16)										
	memptr16	(SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(IP)+(memptr16)										
	far-label	(SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←seg, (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←offset										
	memptr32	(SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(memptr32+2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(memptr32)										
CBW		if (AL)<80H then (AH)←0 else (AH)←FFH	byteを符号付きでwordに変換									
CLC		(CF) ←0	キャリフラグをクリア									0
CLD		(DF) ←0	方向フラグをリセット		0							
CLI		(IF) ←0	割り込み許可フラグをクリア			0						
CMC		(CF) ←$\overline{(CF)}$	キャリフラグを反転									X
CMP	reg,reg	(reg)-(reg)	減算をしてデータ比較	X				X	X	X	X	X
	reg,mem	(reg)-(mem)										
	mem,reg	(mem)-(reg)										
	reg,imm	(reg)-imm										
	mem,imm	(mem)-imm										
	acc,imm	if W=0 (AL)-imm else W=1 (AX)-imm										
CMPS		((SI))-((DI)),　(SI)←(SI)±delta,　(DI)←(DI)±delta	メモリ上の文字列の比較	X				X	X	X	X	X
CWD		if (AX)<8000H then (DX)←0 else (DX)←FFFFH	wordを符号付きでdwordに変換									
DAA		if ((AL) & 0FH)>9 or (AF)=1 then (AL)←(AL)+6,(CF)←(AF)∨(CF) ,(AF)←1 if ((AL) >9FH　or (CF)=1 then (AL)←(AL)+60H, (CF)←1	10進加算後の補正	U				X	X	X	X	X
DAS		if((AL) & 0FH)>9 or (AF)=1 then (AL)←(AL)-6, ,(CF)←(AF)∨(CF) ,(AF)←1 if((AL) >9FH　or (CF)=1 then (AL)←(AL)-60H, (CF)←1	10進減算後の補正	U				X	X	X	X	X
DEC	reg8	(reg)←(reg8)-1	ディクリメント	X				X	X	X	X	
	mem	(mem)←(mem)-1										
	reg16	(reg16)←(reg16)-1										
DIV	reg8	(AL)←(AX)/(reg8),　(AH)←(AX)%(reg8)	符号のない除算	U				U	U	U	U	U
	mem8	(AL)←(AX)/(mem8),　(AH)←(AX)%(mem8)										
	reg16	(AX)←(DX:AX)/(reg16),　(DX)←(DX:AX)%(reg16)										
	mem16	(AX)←(DX:AX)/(mem16),　(DX)←(DX:AX)%(mem16)										
ESC	ext-op,reg	CPU escape, data bus←(reg)	他のCPUへ命令を与える									
	est-op,mem	CPU escape, data bus←(mem)										
HLT		CPU halt	CPUを停止									
IDIV	reg8	(AL)←(AX)/(reg8),　(AH)←(AX)%(reg8)	符号付き乗算	U				U	U	U	U	U
	mem8	(AL)←(AX)/(mem8),　(AH)←(AX)%(mem8)										
	reg16	(AX)←(DX:AX)/(reg16),　(DX)←(DX:AX)%(reg16)										
	mem16	(AX)←(DX:AX)/(mem16),　(DX)←(DX:AX)%(mem16)										
IMUL	reg8	(AX)←(AL)*(reg8)　EXT=AH, LOW=AL	符号付き乗算	X				U	U	U	U	X
	mem8	(AX)←(AL)*(mem8)　EXT=AH,LOW=AL										
	reg16	(DX:AX)←(AX)*(reg16)　EXT=DX,LOW=AX										

命令	オペランド	動作内容	機能	O	D	I	T	S	Z	A	P	C
IMUL	mem16	(DX:AX)←(AX)*(mem16)　EXT=DX,LOW=AX	符号付き乗算									
IN	acc,imm8	if W=0 (AL)←(imm8) else W=1 (AX)←(imm8+1:imm8)	データ入力									
	acc,DX	if W=0 (AL)←((DX)) else W=1 (AX)←((DX)+1:(DX))										
INC	reg8	(reg)←(reg8)+1	インクリメント	X				X	X	X	X	
	mem	(mem)←(mem)+1										
	reg16	(reg16)←(reg16)+1										
INT	3	(SP)←(SP)-2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(14) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(12)	割り込み			0	0					
	im8	(SP)←(SP)-2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(type×4+2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(type×4)										
INTO		if (OF)=1 then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS, (IF)←0,(TF)←0, (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(12H), (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(10H)	オーバフロー割り込み			0	0					
IRET		(IP)←((SP)+1:(SP)), (SP)←(SP)+2, (CS)←((SP)+1:(SP)), (SP)←(SP)+2, FLAGS←((SP)+1:(SP)), (SP)←(SP)+2	割り込み処理からの復帰	R	R	R	R	R	R	R	R	R
JA/JNBE	short-label	if (ZF)=1 (IP)←(IP)+disp	無符号データで、>なら分岐									
JAE/JNB	short-label	if (CF)=0 (IP)←(IP)+disp	無符号データで、>=なら分岐									
JB/JNAE	short-label	if (CF)=1 (IP)←(IP)+disp	無符号データで、<なら分岐									
JBE/JNA	short-label	if (CF)∨(ZF)=1 (IP)←(IP)+disp	無符号データで、<=なら分岐									
JC	short-label	if (CF)=1 (IP)←(IP)+disp	桁上げがあれば分岐									
JCXZ	short-label	if (CX)=0 (IP)←(IP)+disp	CXが0なら分岐									
JE/JZ	short-label	if (ZF)=1 (IP)←(IP)+disp	等しいなら分岐									
JG/JNLE	short-label	if ((SF)∀(OF) ∨(ZF)=0 (IP)←(IP)+disp	有符号データで、>なら分岐									
JGE/JNL	short-label	if (SF)∀(OF)=0 (IP)←(IP)+disp	有符号データで、>=なら分岐									
JL/JNGE	short-label	if (SF)∀(OF)=1 (IP)←(IP)+disp	有符号データで、<なら分岐									
JLE/JNG	short-label	if [(SF)∀(OF) ∨(ZF)=1 (IP)←(IP)+disp	有符号データで、<=なら分岐									
JMP	near-label	(IP)←(IP)+disp	無条件分岐									
	short-label	(IP)←(IP)+disp										
	regptr16	(IP)←(IP)+(regptr16)										
	memptr16	(IP)←(IP)+(memptr16)										
JMP	far-label	(SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←seg, (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←offset	無条件分岐									
	memptr32	(SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(memptr32+2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(memptr32)										
JNC		if (CF)=1 (IP)←(IP)+disp	桁上げがなければ分岐									
JNE/JNZ		if (ZF)=1 (IP)←(IP)+disp	等しくないなら分岐									
JNO		if (OF)=0 (IP)←(IP)+disp	オーバフローがなければ分岐									
JNP/JPO		if (PF)=0 (IP)←(IP)+disp	奇数パリティでなければ分岐									
JNS		if (SF)=0 (IP)←(IP)+disp	負でなければ分岐									
JO		if (OF)=0 (IP)←(IP)+disp	オーバフローならば分岐									
JP/JPE		if (PF)=1 (IP)←(IP)+disp	偶数パリティならば分岐									
JS		if (SF)=1 (IP)←(IP)+disp	負であれば分岐									
LAHF		(AH)←(SF):(ZF):X:(AF):X:(PF):X:(CF)	スタックからAHへフラグを復現									
LDS	reg16,mem32	(reg16)←(mem32)，(DS)←(mem32+2)	ポインタアドレス（DS）のロード									
LEA	reg16,mem16	(reg16) ←mem16	実行アドレスのロード									
LES	reg16,mem32	(reg16)←(mem32)，(ES)←(mem32+2)	ポインタアドレス（ES）のロード									
LOCK		Bus Lock Prefix	バスを次の命令までロックする									
LODS		((acc))←((SI)), (SI)←(SI)±delta	文字列をロード									
LOOP		(CX)←(CX)-1, if (CX)≠0 (IP)←(IP)+disp	ループ									
LOOPE /LOOPZ		(CX)←(CX)-1, if (ZF)=1 and (CX)≠0 (IP)←(IP)+disp	等しければループ									
LOOPNE /LOOPNZ		(CX)←(CX)-1, if (ZF)=0 and (CX)≠0 (IP)←(IP)+disp	等しくなければループ									
MOV	reg,reg	(reg) ←(reg)	データ転送									
	mem,reg	(mem) ←(reg)										
	reg,mem	(reg) ←(mem)										
	mem,imm	(mem) ←imm										
	reg,imm	(reg) ←imm										
	acc,mem	(acc) ←(mem)										
	mem,acc	(mem) ←(acc)										
	sreg,reg16	(sreg) ←(reg16)										
	sreg,mem	(sreg) ←(mem)										
	reg16,sreg	(reg16) ←(sreg)										
	mem,sreg	(mem) ←(sreg)										

命令	オペランド	動作内容	機能	O	D	I	T	S	Z	A	P	C
MOVS		((DI))←((SI)), (SI)←(SI)±delta, (DI)←(DI)±delta	文字列移動									
MOVSB		((DI))←((SI)), (SI)←(SI)±1, (DI)←(DI)±1	文字列をbyte単位で移動									
MOVSW		((DI))←((SI)), (SI)←(SI)±2, (DI)←(DI)±2	文字列をword単位で移動									
MUL	reg8	(AX)←(AL)・(reg8)	符号のない乗算	X				U	U	U	U	X
	mem8	(AX)←(AL)・(mem8)										
	reg16	(DX:AX)←(AX)・(reg16)										
	mem16	(DX:AX)←(AX)・(mem16)										
NEG	reg	(reg) ← 0-(reg)	符号変換	X				X	X	X	X	X
	mem	(mem) ← 0-(mem)										
NOP		no operation	なにも実行しない									
NOT	reg8	(reg8)←FFH-(reg8)										
	mem8	(mem8)←FFH-(mem8)										
	reg16	(reg16)←FFFFH-(mem16)										
	mem16	(mem16)←FFFFH-(mem16)										
OR	reg,reg	(reg) ←(reg)∨(reg)	論理和	0				X	X	U	X	0
	reg,mem	(reg) ←(reg)∨(mem)										
	mem,reg	(mem) ←(mem)∨(reg)										
	reg,imm	(reg) ←(reg)∨imm										
	mem,imm	(mem) ←(mem)∨imm										
	acc,imm	if W=0 (AL)←(AL)∨imm else W=1 (AX)←(AX)∨imm										
OUT	imm8,acc	if W=0 (imm8)←(AL) else W=1 (imm8+1:imm8)←(AL)	データ出力									
	DX,acc	if W=0 ((DX))←(AL) else W=1 ((DX)+1:(DX))←(AX)										
POP	mem	(mem) ←((SP)+1:(SP))←(mem), (SP) ←(SP)+2	スタックから復元									
	reg	(mem) ←((SP)+1:(SP))←(reg), (SP) ←(SP)+2										
	sreg	(mem) ←((SP)+1:(SP))←(sreg), (SP) ←(SP)+2										
POPF		FLAGS ←((SP)+1:(SP)), (SP) ←(SP)+2	フラグをスタックから復元	R	R	R	R	R	R	R	R	R
PUSH	mem	(SP) ←(SP)-2, ((SP)+1:(SP))←(mem)	スタックへ保存									
	reg	(SP) ←(SP)-2, ((SP)+1:(SP))←(reg)										
	sreg	(SP) ←(SP)-2, ((SP)+1:(SP))←(sreg)										
PUSHF		(SP) ←(SP)-2, ((SP)+1:(SP))←FLAGS	フラグをスタックへ保存									
RCL	RCL reg,1	←(CF)←(reg)← 1bit左回転	キャリフラグを含めた	X								X
	RCL reg,CL	←(CF)←(reg)← CLの値だけ左回転	左ローテート	U								X
	RCL mem,1	←(CF)←(mem)← 1bit左回転		X								X
	RCL mem,CL	←(CF)←(mem)← CLの値だけ左回転		U								X
RCR	RCR reg,1	→(CF)→(reg)→ 1bit右回転	キャリフラグを含めた	X								X
	RCR reg,CL	→(CF)→(reg)→ (CL)の値bit、右回転	右ローテート	U								X
	RCR mem,1	→(CF)→(mem)→ 1bit右回転		X								X
	RCR mem,CL	→(CF)→(mem)→ (CL)の値bit、右回転		U								X
REP		(CX)≠0であれば、 続くストリング命令を実行,(CX)←(CX)-1	ストリング命令を繰り返す									
REPE /REPZ		(CX)≠0で(ZF)=1の時、 続くストリング命令を実行,(CX)←(CX)-1	ストリングの比較結果が 等しいなら繰り返す									
REPNE /REPNZ		(CX)≠0で(ZF)=0の時、 続くストリング命令を実行,(CX)←(CX)-1	ストリングの比較結果が 等しくないなら繰り返す									
RET		(IP)←((SP)+1:(SP)),(SP)←(SP)+2	手続きからの復帰									
	pop-value	(IP)←((SP)+1:(SP)),(SP)←(SP)+data										
		(IP)←((SP)+1:(SP)),(SP)←(SP)+2, (CS)←((SP)+1:(SP)),(SP)←(SP)+2										
	pop-value	(IP)←((SP)+1:(SP)),(SP)←(SP)+2, (CS)←((SP)+1:(SP)),(SP)←(SP)+data										

命令	オペランド	動作内容	機能	O	D	I	T	S	Z	A	P	C
ROL	reg,1	(CF)←(reg)← 1bit左回転	左ローテート									
	reg,CL	(CF)←(reg)← (CL)の値bit、左回転										
	mem,1	(CF)←(mem)← 1bit左回転										
	mem,CL	(CF)←(mem)← (CL)の値bit、左回転										
ROR	reg,1	(CF)→(reg)→ 1bit右回転	右ローテート									
	reg,CL	(CF)→(reg)→ (CL)の値bit、右回転										
	mem,1	(CF)→(mem)→ 1bit右回転										
	mem,CL	(CF)→(mem)→ (CL)の値bit、右回転										
SAHF		(SF):(ZF):X:(AF):X:(PF):X:(CF)←(AH)	スタックへAHからフラグを保存					R	R	R	R	R
SAL /SHL	reg,1	(CF)←(reg)←0 1bit左移動	算術左シフト	X				X	X	U	X	X
	mem,1	(CF)←(mem)←0 1bit左移動										
	reg,CL	(CF)←(reg)←0 (CL)の値bit、左移動										
	mem,CL	(CF)←(mem)←0 (CL)の値bit、左移動										
SAR	reg,1	(LSB of reg) →(reg)→(CF) 1bit右移動	算術右シフト	X				X	X	U	X	X
	mem,1	(LSB of mem) →(mem)→(CF) 1bit、右移動										
	reg,CL	(LSB of reg) →(reg)→(CF) (CL)の値bit、右移動										
	mem,CL	(LSB of mem) →(mem)→(CF) (CL)の値bit、右移動										
SBB	reg,reg	(reg) ←(reg)-(reg)-(CF)	借りを含めた減算	X				X	X	X	X	X
	reg,mem	(reg) ←(reg)-(mem)-(CF)										
SBB	mem,reg	(mem) ←(mem)-(reg)-(CF)	借りを含めた減算									
	reg,imm	(reg) ←(reg)-imm-(CF)										
	mem,imm	(mem) ←(mem)-imm-(CF)										
	acc,imm	if W=0 (AL)←(AL)-imm-(CF) else W=1 (AX)←(AX)-imm-(CF)										
SCAS		(DI)-(SI), (SI)←(SI)±delta, (DI)←(DI)±delta	accとメモリ上の文字列比較	X				X	X	X	X	X
SHL /SAL	reg,1	(CF)←(reg)←0 1bit、左移動	左シフト	X				X	X	U	X	X
	mem,1	(CF)←(mem)←0 1bit、左移動										
	reg,CL	(CF)←(reg)←0 (CL)の値bit、左移動										
	mem,CL	(CF)←(mem)←0 (CL)の値bit、左移動										
SHR	reg,1	0 →(reg)→(CF) 1bit、右移動	右シフト	X				X	X	U	X	X
	mem,1	0 →(mem)→(CF) 1bit、右移動										
	reg,CL	0 →(reg)→(CF) (CL)の値bit、右移動										
	mem,CL	0 →(mem)→(CF) (CL)の値bit、右移動										
STC		(CF)←1	キャリフラグをセット									1
STD		(DF)←1	方向フラグをセット		1							
STI		(IF)←0	割り込み許可フラグをセット			1						
STOS		((DI))←((acc)), (DI)←(DI)±delta	文字列をストア									
SUB	reg,reg	(reg) ←(reg)-(reg)	減算	X				X	X	X	X	X
	reg,mem	(reg) ←(reg)-(mem)										
	mem,reg	(mem) ←(mem)-(reg)										
	reg,imm	(reg) ←(reg)-imm										
	mem,imm	(mem) ←(mem)-imm										
	acc,imm	if W=0 (AL)←(AL)-imm else W=1 (AX)←(AX)-imm										
TEST	reg,reg	(reg) ∧ (reg)	論理積をとって比較	0				X	X	U	X	0
	reg,mem	(reg) ∧ (mem)										
	mem,reg	(mem) ∧ (reg)										
	reg,imm	(reg) ∧ imm										
	mem,imm	(mem) ∧ imm										
	acc,imm	if W=0 (AL) ∧ imm else W=1 (AX) ∧ imm										
WAIT		CPU wait	TESTピンがアクティブまで待つ									
XCHG	reg,reg	(reg)←→(reg)	データ交換									
	mem,reg	(mem)←→(reg)										
	AX,reg16	(AX)←→(reg16)										

命令	オペランド	動作内容	機能	O	D	I	T	S	Z	A	P	C
XLAT		(AL)←((BX)+(AL))	表データ転送									
XOR	reg,reg	(reg) ←(reg) ∀(reg)	排他的論理和	0				X	X	U	X	0
	reg,mem	(reg) ←(reg) ∀(mem)										
	mem,reg	(mem) ←(mem) ∀(reg)										
	reg,imm	(reg) ←(reg) ∀imm										
	mem,imm	(mem) ←(mem) ∀imm										
	acc,imm	if W=0 (AL)←(AL) ∀imm else W=1 (AX)←(AX) ∀imm										

reg	8/16bit汎用レジスタ
reg8	8bit汎用レジスタ
reg16	16bit汎用レジスタ
mem	8/16bitメモリ番地
mem8	8bitメモリ番地
mem16	16bitメモリ番地
mem32	32bitメモリ番地
imm	0-FFFFHの定数
imm8	0-FFHの定数
acc	AXまたはALレジスタ
sreg	セグメントレジスタ

near-proc	現コードセグメント内の手続き
far-proc	他のコードセグメント内の手続き
short-label	現コードセグメント内のラベル
near-label	現コードセグメント内のラベル
far-label	他のコードセグメント内のラベル
memptr16	現コードセグメント内のオフセットを持つword
memptr32	他のコードセグメント内のオフセットとセグメントを含むdouble word
regptr16	現コードセグメント内のオフセットを持つwordレジスタ
pop-value	スタックから捨てるbyte数
delta	インデックスレジスタの増減値 W=0 の時±1 、W=1 の時、±2 DF=0 の時＋ 、DF=1 の時ー
W	ワード／バイトフィールド

←	転送方向
＋	加算
ー	減算
・	乗算
／	除算
%	余り
∧	論理積
∨	論理和
∀	排他的論理和

R	復帰
X	変化
U	未定義

( ) レジスタまたはメモリの内容
(( )) レジスタまたはメモリが指すメモリの内容

付録 2 MS-DOSファンクション一覧(抜粋)

番号	機能	呼び出し	戻り値
01	キーボード入力（エコー付き）	AH←01	AL　入力文字
02	1文字出力	AH←02 DL←出力文字	
05	プリンタ出力	AH←05	
06	コンソール直接入出力　入力	AH←06 DL←FF	AL　入力文字
	出力	AH←06 DL←出力文字	
07	コンソール直接入力	AH←07	AL　入力文字
08	キーボード入力（エコーなし）	AH←08	AL　入力文字
09	文字列出力	AH←09 DX←文字列の先頭アドレス	
0A	バッファへのキーボード入力	AH←0A DX←バッファの先頭アドレス	
0B	キーボードのステータスチェック	AH←0B	AL　キーボードステータス
0C	バッファを空にしてキーボード入力	AH←0C DX←バッファの先頭アドレス	
2A	日付の読み出し	AH←2A	CX　年 DH　月 DL　日 AL　曜
2B	日付の設定	AH←2B DH←　月 DL←　日 AL←　曜	AL=00　正常終了 AL=FF　無効な日付
2C	時刻の読み出し	AH←2C	CH　時 CL　分 DH　秒 DL　1/100秒
2D	時刻の設定	AH←2D CH←時 CL←分 DH←秒 DL←1/100秒	AL=00　正常終了 AL=FF　無効な時刻
30	MS-DOSバージョンの読み出し	AH←30	AH　バージョン小数部 AL　バージョン整数部
38	国別情報の読み出し	AH←38	
4C	プロセスの終了	AH←4C AL←リターンコード	
5E	マシン名の取得	AH←5E AL←00 DX←マシン名バッファ(16バイト) の先頭番地	

付録 3 MS-DOSコマンド一覧(抜粋)

種類	コマンド	機能	使用法	
内部コマンド	cd	カレントディレクトリの変更	cd dir1 cd ¥ cd . .	dir1へ移る ルートディレクトリへ移る 上位ディレクトリへ戻る
	copy	ファイルのコピー	copy file1 file2	file1をfile2へコピー
	debug	実行ファイルのデバッグ	debug mov.exe	mov.exeをデバッグ
	del	ファイルの削除	del file1	file1を削除
	dir	ディレクトリ内容の表示	dir dir *.asm	全ファイルを表示 属性.Asmファイルの表示
	md	ディレクトリの作成	md newdir	ディレクトリnewdirを作成
	path	コマンドを探索する ディレクトリの設定	path ¥masm path %path%;¥masm path prg	パスをルート下のmasmへ 変更 現パスに、¥masmを加える パスをprgに設定
	ren	ファイル名を変更	ren oldfile newfile	ファイル名oldfileをnewfile へ変更
	rd	ディレクトリの削除	rd olddir	ディレクトリolddirを削除
	type	ファイル内容の表示	type file.asm	ファイルfile.asmの内容を 表示
外部コマンド	ml	マイクロソフトMASM(DDK) によるアセンブル	ml exmpl.asm	exmpl.asmをアセンブル
	link	リンカの起動	link exmple iolib	exmple.objとiolib.objをリン クし、exmple.exeを生成.

付録 4 Microsoft DDKについて

マイクロソフト社のマクロアセンブラMASMを含んだ開発用キットのDDKは，下に示すマイクロソフト社のホームページ（平成14年1月16日現在）

http://www.microsoft.com/ddk/

から，ライセンスについての承認を受ければダウンロードすることができます．DDKをインストールするためには次の2つをダウンロードして，解凍します．

BINS_DDK.EXE

98SETUP.EXE

これにより，ml.exe，ml.err，link.exeなどを，たとえば¥masm32¥binディレクトリに生成します．

ここで生成されるlink.exeは，32bitリンカのため，本書のプログラム例などのリアルモードプログラム(16bit プログラム)を作るときは，

ftp://ftp.microsoft.com/softlib/mslfiles/lnk563.exe

を解凍しlink.exeを生成します．このlink.exeをmasm¥binディレクトリへコピーして，32ビットのリンカと交換します．32ビットのlink.exeが必要なら適当に名前を変更して保存しておきます．

■MS-DOS上でのアセンブルとリンク

アセンブル　**ml /c /Zm /Fl /Fm exmpl.asm**

リンク　**link exmpl**

■QEDITOR上でアセンブル，リンク

masm32¥binディレクトリ内のすべてのバッチファイルの変更が必要になります．xxx.batとある全バッチファイル内にあるmlオプションとlinkオプションを次のように変更します．

```
\ml  /c  /coff
      ▼
\ml   /c   /Zm  /Fl  /Fm

\masm32\bin\Link  /SUBSYSTEM:WINDOWS  %1.obj  rsrc.obj
      ▼
\masm32\bin\Link   %1.obj  rsrc.obj
```

これにより，QEDITORからのアセンブル，リンク，実行が可能となります．

付録

付録 5 DDK MASM オプション

使用法：

ML [/options] filelist [/link linkoptions]

オプション：

/AT Enable tiny model (.COM file)
/Bl<linker> Use alternate linker
/c Assemble without linking
/Cp Preserve case of user identifiers
/Cu Map all identifiers to upper case
/Cx Preserve case in publics, externs
/coff generate COFF format object file
/D<name>[=text] Define text macro
/EP Output preprocessed listing to stdout
/F <hex> Set stack size (bytes)
/Fe<file> Name executable
/Fl[file] Generate listing
/Fm[file] Generate map
/Fo<file> Name object file
/FPi Generate 80x87 emulator encoding
/Fr[file] Generate limited browser info
/FR[file] Generate full browser info
/G<c|d|z> Use Pascal, C, or Stdcall calls
/H<number> Set max external name length
/I<name> Add include path
/link <linker options and libraries>
/nologo Suppress copyright message
/Sa Maximize source listing
/Sc Generate timings in listing
/Sf Generate first pass listing
/Sl<width> Set line width
/Sn Suppress symbol-table listing
/Sp<length> Set page length
/Ss<string> Set subtitle
/St<string> Set title
/Sx List false conditionals
/Ta<file> Assemble non-.ASM file
/w Same as /W0 /WX
/WX Treat warnings as errors
/W<number> Set warning level
/X Ignore INCLUDE environment path
/Zd Add line number debug info
/Zf Make all symbols public
/Zi Add symbolic debug info
/Zm Enable MASM 5.10 compatibility
/Zp[n] Set structure alignment
/Zs Perform syntax check only

付録6 Microsoft リアルモードLinker Version 5.60 オプション

使用法:

LINK

LINK @<response file>

LINK <objs>,<exefile>,<mapfile>,<libs>,<deffile>

オプション:

/?
/ALIGNMENT
/BATCH
/CODEVIEW
/CPARMAXALLOC
/DOSSEG
/DSALLOCATE
/DYNAMIC
/EXEPACK
/FARCALLTRANSLATION
/HELP
/HIGH
/INFORMATION
/LINENUMBERS
/MAP
/NODEFAULTLIBRARYSEARCH
/NOEXTDICTIONARY
/NOFARCALLTRANSLATION
/NOGROUPASSOCIATION
/NOIGNORECASE
/NOLOGO
/NONULLSDOSSEG
/NOPACKCODE
/NOPACKFUNCTIONS
/NOFREEMEM
/OLDOVERLAY
/ONERROR
/OVERLAYINTERRUPT
/PACKCODE
/PACKDATA
/PACKFUNCTIONS
/PAUSE
/PCOD
/PMTYPE
/QUICKLIBRARY
/SEGMENTS
/STACK
/TINY
/WARNFIXUP

付録 7 多摩ソフトウェア社 Light Macro Assemblerについて

多摩ソフトウェア社のLight Macro Assemblerは，マイクロソフト社のMASM6.0と互換性のあるマクロアセンブラで，MS-DOS上で動作します．Light Macro Assemblerにはリンカ，ライブラリマネージャ，エディタが付属しており，簡単に使用できるデバッガも備えています．

Light Macro Assemblerは多摩ソフトウェア社のホームページから体験版をダウンロードすることができます．体験版は100行未満のプログラムしかアセンブルできませんが，本書のプログラムならば利用できる範囲です．

次に示す多摩ソフトウェア社のホームページから入手できます．(平成14年2月現在).

http://www.tamasoft.co.jp/lasm/index.html

付録 8 MASM疑似命令一覧(抜粋)

疑似命令	使用法	機能
.386	.386	80386非特権命令をアセンブル可能にする
.8086	.8086	8086命令をアセンブル可能にする.上位プロセサ命令は禁止
.8087	.8087	8087コプロセッサ命令をアセンブル可能にする.上位コプロセッサ命令は禁止
.code	.CODE	コードセグメントnameの始まり
.const	.CONST	定数データセグメントの始まり
.data	.DATA	初期化データセグメントの始まり
.data?	.DATA?	未初期化データセグメントの始まり
.model	.MODEL memorymodel	メモリモデルをmemorymodelに指定 small,compact,medinum,large,hugeなど
.stack	.STAC	スタックセグメントの始まり
assume	ASSUME	セグメントレジスタsegregとセグメントnameを対応付ける
comment	COMMENT delimiter [text]	delimiterで囲まれたすべてのtextをコメント扱いする
db	[name] DB initial [,initial]	byteデータ領域nameに,初期値initialを割り当て
dd	[name] DD initial [,initial]	double word(4byte)データ領域nameに,初期値initialを割り当てる
df	[name] DF initial [,initial]	far word(6byte)データ領域nameに,初期値initialを割り当てる
dosseg	DOSSEG	MS-DOSのセグメント順序によってセグメントを並べ替える
dq	[name] DQ initial [,initial]	quad word(8byte)データ領域nameに,初期値initialを割り当てる
dt	[name] DT initial [,initial]	10byteデータ領域nameに,初期値initialを割り当てる
dw	[name] DW initial [,initial]	word(2byte)データ領域nameに,初期値initialを割り当てる
else	ELSE	条件が偽のときアセンブルされるブロックの始まり
elseif	ELSEI expression	前条件が偽で,条件expressionが真ならアセンブルされるブロックの始まり
end	END[startaddress]	モジュールの終わり,startaddressからのプログラム実行を指定
endif	ENDIF	条件ブロックの終わり

endm	ENDM	マクロブロックの終わり
endp	name ENDP	プロシージャnameの終わり
ends	name ENDS	セグメントnameの終わり
equ	name EQU expression	名称nameに,式expressionを割り当て
even	EVEN	次の命令,変数を偶数バイト境界に合わせる
extrn	EXTRN name:type	外部変数,ラベルnameのタイプをの定義
.fardata	.FARDATA[name]	初期化済みfarデータセグメントnameの始まり
group	name GROUP segment	グループ名nameとセグメントsegmentを対応づける
if	IF expression	条件expressionが真ならアセンブル
ifdef	IFDEF name	変数またはシンボルnameが既に定義されているならアセンブル
ife	IFE expression	expressionが偽(0)ならアセンブル
include	INCLUDE filename	ソースファイルfilenameを,アセンブル中のソースファイルに挿入
label	name LABEL type	ラベルnameをtype型として定義
local	LOCAL vardef	スタック上にローカル変数vardefを割り当てる
macro	name MACRO param[,param]	マクロブロックnameの始まり.引数param,・・・・
name	NAME modulename	モジュール名modulenameを設定する
org	ORG expression	ロケーションカウンタをexprssionに設定する
page	PAGE [[length,]width]	プログラムリストをlength行,width幅に設定する
proc	name PROC[near¦far]	プロシージャnameの始まり
public	PUBLIC name[,name]	変数,ラベルnameを他のプロシージャから参照可能にする
record	name RECORD field[,field]	フィールドfieldを持つレコードnameを定義
segment	name SEGMENT	セグメントnameの始まり
struct	name STRUCT field[,field]	フィールドfieldを持つ構造体nameを定義
subttl	SUBTTL text	プログラムのサブタイトルtextを定義
title	TITLE text	プログラムのタイトルtextを定義

[ ]はオプション

付録 9 MASM演算子一覧(抜粋)

演算子	使用法	機能
+,-,*,/,mod	expression1 + expression2	expression1とexpression2の算術値をとる
and,or,not,xor	expression1 AND expression2	expression1とexpression2をビット対応で論理値をとる
dup	count DUP initialvalue	count個の初期値initialvalueの指定
eq,ge,gt,le,lt,ne	expression1 EQ expression2	expression1とexpression2を論理比較する.真-1,偽0
high	HIGH expression	expressionの上位バイトを返す
length	LENGTH name	シンボルnameの要素数を返す
low	LOW expression	expressionの下位バイトを返す
offset	OFFSET expression	expressionのオフセット値を返す
ptr	type PTR expression	expressionをtype型に変換して取り扱う
shl,shr	expression1 SHR expression2	expression1をexpression2ビット論理シフトする
short	SHORT label	ラベルlabelの型をショートに設定
size	SIZE name	シンボルnameがdup演算子で定義されているとき,そのバイト数を返す
type	TYPE expression	expressionの型を返す
width	WIDTH name	レコードnameのフィールド長をビット数で返す

付録 10 デバッグ

プログラムが意図したとおりの働きをしない場合に，そのバグを見つけるために，デバッガを用います．現在入手できる16ビットコードのデバッガには，Turbo Debugger，Lihgt Macro Assemblerのデバッガ，MS-DOS上の機械語デバッガdebugなどがあります．

ここでは，MS-DOS上の機械語デバッガdebugについて説明します．debugではブレイクポイントを設定して，プログラムを実行することができます．debugが起動した際のコマンドを，表に示します．debugは機械語についてのデバッガであるため，コマンドの実行には充分注意する必要があります．特に入出力に関するコマンドやEMSメモリに関するコマンドは用いないほうが安全でしょう．これらのコマンドには，使用例を記さないことにします．また，debugコマンドでデバッグする際には，コード，データ，スタックセグメントをきちんと定義する必要があります．

実際に3章の**リスト3．2**に示したプログラムをデバッグする例を示します．スタックセグメントを定義していないため，プログラムに

```
.stack 100H
```

を付け加えてアセンブルとリンクをして，実行形式プログラムmov.exeを作ります．この.stack疑似命令が加えられていないと，思わぬ動作をすることがありますから，注意が必要です．

MS-DOS上でdebugコマンドでmov.exeを指定して起動すると，機械語プログラムがCS:上にロードされます．

最初のuコマンドで，コードセグメントCS:の0～15H番地の手前までを表示しています．最初の2つのmov命令で，2311がDSレジスタに代入されており，このmov.exeプログラムの実行時には，データセグメントの先頭が23110H番地であることがわかります．データセグメントの番地は，debugコマンドによって決められるため，いつもこの番地になるとは限らないので注意が必要です．

次のrコマンドで，レジスタの内容を表示します．CSレジスタの値から，コードセグメントは23100H番地から始まることがわかります．DSレジスタの値はプログラム実行時に与えられるため，ここで表示されている値はdebugプロ

グラム側の値になっています．

次のdコマンドで，実行時のデータセグメント（2311:）の0～F番地の内容を表示します．データセグメントの番地は，すでに変数x(8番地)に2A('＊')が与えられていることがわかります．

次のgコマンドで，CS:の5番地にブレイクポイントを設定して，0番地から実行します．実行後レジスタの内容が表示され，インストラクションポインタIPが，ブレイクポイントの05番地を指しています．

次のgコマンドで，CS:の13番地にブレイクポイントを設定して，5番地から実行します．実行後レジスタの内容が表示され，インストラクションポインタIPが，ブレイクポイントの13番地を指しています．

次のdコマンドで，データセグメントの0～F番地の内容を表示します．ここで変数y(9番地)に'＊'が転送されていることが示されています．

C:¥prg>debug mov.exe　**MS-DOS　debugコマンドでmov.exeをデバッグ**

-u cs:0 16　**CS:の0-15H番地を逆アセンブル　表示されたmov.exeプログラム**

2310:0000 B81123	MOV	AX,2311	**DS:の先頭は23110H番地**
2310:0003 8ED8	MOV	,AX	
2310:0005 A00800	MOV	L,[0008]	**変数xはDS:08番地**
2310:0008 A20900	MOV	[0009],AL	**変数yはDS:09番地**
2310:000B 8A160900	MOV	DL,[0009]	
2310:000F B402	MOV	AH,02	
2310:0011 CD21	INT	21	
2310:0013 B8004C	MOV	AX,4C00	
2310:0016 CD21	INT	21	

```
-r              レジスタの表示

AX=0000  BX=0000  CX=001A  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=2300  ES=2300  SS=2312  CS=2310  IP=0000   NV UP EI PL NZ NA PO NC
2310:0000 B81123        MOV     AX,2311

-d 2311:0 f     DS:の0～FH番地までを表示                        変数xとy
2311:0000  02 CD 21 B8 00 4C CD 21-2A 00 DE E9 8B 36 F3 DF   ..!..L.!*....6..

-g =cs:0 5      CS:5番地にブレイクポイントを設定して，0番地から実行

AX=2311  BX=0000  CX=001A  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=2311  ES=2300  SS=2312  CS=2310  IP=0005   NV UP EI PL NZ NA PO NC
2310:0005 A00800        MOV     AL,[0008]                   DS:0008=2A

-g =cs:5 13     CS:13番地にブレイクポイントを設定して，5番地から実行
*               プログラムの実行で表示された'*'

AX=022A  BX=0000  CX=001A  DX=002A  SP=0100  BP=0000  SI=0000  DI=0000
DS=2311  ES=2300  SS=2312  CS=2310  IP=0013   NV UP EI PL NZ NA PO NC
2310:0013 B8004C        MOV     AX,4C00

-d 2311:0 f     DS:の0～FH番地までを表示                        変数xとy
2311:0000  02 CD 21 B8 00 4C CD 21-2A 2A DE E9 8B 36 F3 DF   ..!..L.!**...6..

-q              debugの終了
```

デバッグコマンド

コマンド	引数	機能	使用法	
?		ヘルプ	?	全debugコマンドを表示
A	[アドレス]	アセンブル	a 0	0番地をアセンブル
C	範囲 アドレス	比較のため表示	c ds:0 10 100	DS:0～10番地と100番地～を表示
D	[範囲]	メモリダンプ	d ds:0 15	DS:0～15番地をダンプ
E	アドレス [一覧]	入力	／	
F	範囲 一覧	データの書き込み	f ds:0 10 ff	DS:0～10H番地にFFを書き込み
G	[=アドレス] [アドレス]	実行	g =0 5	5番地にbreak pointを設定し、0番地から実行
H	数値1 数値2	加算したアドレスの内容を表示	h ds:10 22	10H+22H番地の内容を表示
I	ポート	入力	／	
L	[アドレス] [ドライブ] [先頭セクタ] [数値]	読み込み	／	
M	範囲 アドレス	転送	m ds:0 10 100	DS:0～10番地のデータを、100～へ転送
N	[パス名] [引数一覧]	名前付け	／	
O	ポート バイト値	出力	／	
P	[=アドレス] [数値]	続行	p	1命令実行し、レジスタ表示
Q		デバッグ終了	q	デバッグの終了
R	[レジスタ]	レジスタ操作	r	レジスタの表示
S	範囲一覧	検索	s	ds:0 15 ff DS:0～15H番地でFFを検索
T	[=アドレス] [数値]	トレース	t	1命令実行し、レジスタ表示
U	[範囲]	逆アセンブル	u 0	15 CS:の0～15H番地を逆アセンブル
W	[アドレス] [ドライブ] [先頭セクタ] [数値]	メモリ書き込み	／	
XA	[ページ数]	EMS メモリの割り当て	／	
XD	[ハンドル]	EMS メモリの割り当て解除	／	
XM	[論理ページ] [物理ページ] [ハンドル]	EMS メモリページのマップ	／	
XS		EMS メモリ状態の表示	／	

付録11 ASCIIコード表

10進	16進	文字	10進	16進	文字	10進	16進	文字	10進	16進	文字
0	00	^@ NUL	32	20	SPC	64	40	@	96	60	`
1	01	^A SOH	33	21	!	65	41	A	97	61	a
2	02	^B STX	34	22	"	66	42	B	98	62	b
3	03	^C ETX	35	23	#	67	43	C	99	63	c
4	04	^D EOT	36	24	$	68	44	D	100	64	d
5	05	^E ENQ	37	25	%	69	45	E	101	65	e
6	06	^F ACK	38	26	&	70	46	F	102	66	f
7	07	^G BEL	39	27	'	71	47	G	103	67	g
8	08	^H BS	40	28	(	72	48	H	104	68	h
9	09	^I HT	41	29	)	73	49	I	105	69	i
10	0A	^J LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P DLE	48	30	0	80	50	P	112	70	p
17	11	^Q DC1	49	31	1	81	51	Q	113	71	q
18	12	^R DC2	50	32	2	82	52	R	114	72	r
19	13	^S DC3	51	33	3	83	53	S	115	73	s
20	14	^T DC4	52	34	4	84	54	T	116	74	t
21	15	^U NAK	53	35	5	85	55	U	117	75	u
22	16	^V SYN	54	36	6	86	56	V	118	76	v
23	17	^W ETB	55	37	7	87	57	W	119	77	w
24	18	^X CAN	56	38	8	88	58	X	120	78	x
25	19	^Y EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^＼ FS	60	3C	<	92	5C	＼	124	7C	¦
29	1D	^] GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^ RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_ US	63	3F	?	95	5F	_	127	7F	DEL

演習問題解答

1章　1章参照.

2章

1.

(1)	①01000001	②01010101	③10101010	④11111111
(2)	①11111111	②11000001	③01111111	④10000000
(3)	①105	②−127	③−16	④−1
(4)	①50	②D3	③E2	④7F

2.

(1)
```
mov bx,ax      8B D8
mov ax,bx      8B C3
```

(2)〜(5)2章参照.

3章

1. 3章参照.

2. 一部を除きセグメント定義は省略.

(1)
```
.code
    mov al,x
.data
    x db ?
```

(2)
```
.code
    mov y,bh
.data
    y db ?
```

(3)
```
mov al,x
mov y,al
```

(4)
```
mov ax,w
mov w,ax
```

(5)
```
mov h,ah
mov l,al
```

(6)
```
mov bx,w
```

(7)
```
xchg a,ax
xchg b,ax
xchg a,ax
```

(8)
```
mov ax,1000h
mov ds,ax
```

(9)

```
    mov ax,x
    mov bx,y
    mov cx,z
    mov x,cx
    mov y,ax
    mov z,bx
```

xchg命令なら

```
    xchg ax,x
    xchg ax,y
    xchg ax,z
    xchg ax,x
```

(10)

```
lea si,x
lea di,y
mov al,[si]
mov [di],al
```

(11)

```
push a
push b
pop bx
pop ax
```

3. 3章参照.

4章

1. 一部を除きセグメント定義を省略.

(1)

```
 .code
    mov al,x
    add al,y
    mov a,al
  .data
    x db 1
    y db 2
```

(2)

```
mov ax,m
sub ax,n
mov a,ax
```

(3)

```
mov ax,x
lea si,x
add ax,[si]
mov a,ax
```

(4)

```
mov ax,x
add al,byte ptr y
adc ah,byte ptr y+1
mov a,ax
```

(5)

```
mov ax,x
sub al,byte ptr y
sbb ah,byte ptr y+1
mov a,ax
```

(6)

```
dec b
mov al,b
mov a,al
```

(7)

```
inc w
mov ax,w
mov a,ax
```

(8)

```
mov al,b
xor ah,ah
mul w
mov h,dx
mov l,ax
```

(9)

```
mov ax,a
cbd
idiv b
mov x,ax
mov y,dx
```

(10)

```
mov al,b
mov bl,5
imul bl
mov w,ax
```

(11)

```
mov al,y
mov bl,10
mul bl
add al,x
mov w,ax
```

(12)

```
.code
    mov al,x
    xor ah,ah
    add al,y
    aaa
    mov a,ax
.data
    x db '7'
    y db '5'
    a dw ?
```

(13)

```
mov al,x
xor ah,ah
sub al,y
mov a,al
```

(14)

```
mov al,x
mul al,y
aam
mov a,ax
```

2. 4章参照.

5章

1. 一部を除きセグメント定義を省略.

(1)

```
.code
    xor al,al
    mov x,al
.data
    x  db ?
```

(2)

```
.code
    not w
.data
    w  dw 0ffH
```

(3)

```
mov al,x
and al,ofH
mov y,al
```

(4)

```
mov ax,x
or ax,y
mov z,ax
```

(5)

```
sar x,1
```

(6)

```
mov cl,4
shl x,cl
```

(7)

```
mov cl,3
shl x,cl
```

(8)

```
mov cl,2
shr x,cl
```

(9)

```
mov cl,y
rol x,cl
```

(10)

```
mov cl,n
lea di,x
ror [di],cl
```

2. 5章参照.

6章

1. 一部を除きセグメント定義とラベルを省略.

(1)

```
.code
    mov al,x
    cmp al,y
    je lbl
     . . . . .
lba:
.data
    x db 2
    y db 3
```

(2)

```
mov al,x
cmp al,y
jb lbb
jmp lba
```

(3)

```
mov al,x
cmp al,y
jge lbg
jmp lbe
```

(4)

```
mov ax,x
sub ax,y
mov sgn,1
js lba
mov sgn,0
```

(5)

```
    mov cx,y
lp: shl x,1
    loop lp

loop lp
```

(6)

```
    mov cx,10
    lea si,x
    mov al,0ffH
lp:
    inc si
    loop lp
```

(7)

```
    mov cx,10
    lea si,x
    xor al,al
mov [si],al
    inc si
    loop lp
```

(8)

```
    mov cx,y
    xor ax,ax
lp: add ax,x
lp: mov [si],al
    mov a,ax
```

(9)

```
     xor cx,cx
     mov bx,y
     mov ax,x
 lp: cmp ax,bx
     jl  lex
     sub ax,bx
     inc cx
     jmp lp
lex: mov a,cx
```

2．3．6章参照．

7章

1．一部を除きセグメント定義とラベルを省略．

(1)

```
 .code
     cld
     lea si,x
     lea di,y
     mov cx,10
     rep movsb
 .data
     x db 'abcdefghij'
     y db 10 dup(?)
```

(2)

```
std
lea si,x+10
lea di,y+10
mov cx,10
rep movsb
```

(3)

```
    cld
    lea si,x
    mov cx,10
    lea bx,tbl
lp: lodsb
    sub al,'a'
    xlat
    mov [si]-1,al
    loop lp
```

(4)

```
.code
     cld
     mov cx,10
     mov n,0
     mov al,x
     lea di,letter
lp:  scasb
     je eq
     inc n
     loop lp
     jmp neq
```

2. 7章参照.

8章

1. 8章参照.
2. セグメント0の12～15番地
3. ・排他処理ができないハードウェアにおける不具合を回避するため.
   ・割り込み処理中に同じ割り込みがかかることにより，メインの処理に戻れなくなることを回避するため.

9章　9章参照.

10章　10章参照.

11章　11章参照.

参考・参照文献

1)石田晴久：’マイクロコンピュータのプログラミング’、岩波書店

2)河西朝雄：’実用マクロアセンブラ’、技術評論社

3)押野崇芳：’8086/16ビットCPUアセンブラ入門’、日刊工業新聞社

4)蒲地輝尚：’はじめて読む8086’、アスキー出版局

5)モース、内藤訳：’8086入門’、システムソフト

6)ラッセル・レクター、ジョージ・アレクシー、吉川敏則訳：’ザ8086ブック’、産報出版

7)井出裕巳：'MCS-86 16ビットマイクロコンピュータ’、インテル

8)知名定清：’第4世代マイクロプロセサ8086’、インテルジャパンニュース、Vol2-2、53.6.1

9)Intel：'The 8086 Family User's Manual '、インテル

10)NEC半導体応用技術本部 ：’マルチチップ”、日本電気

11)Micorsoft：’ Microsoft Macro Assembler マニュアル、マイクロソフト

12)Borland：Turbo Assembler　マニュアル、ボーランド

13)アスキー書籍編集部：’MS-DOSハンドブック’、アスキー

14)マイクロソフトDDKについて：http://www.microsoft.com/ddk

15)多摩ソフトウエア Light Macro Assemblerについて：
http://www.tamasoft.co.jp/lasm/index.html

＊ホームページのURLはすべて平成14年2月現在のものです.

索引

◆英数字◆

10進数 …………22
10進補正命令 …………106
16進数 …………23
1の補数 …………25
2進化10進数 …………106
2進数 …………22
2の補数 …………24
ASCIIコード …………85
BCD …………106
BCDコード …………88
bit …………22
I/O空間 …………9
I/Oポート …………174
last in-first out …………68
LSB …………22
MSB …………22
MS-DOSシステムコール …………196
NMI …………177

◆あ行◆

アーキテクチャ …………8
アキュムレータ …………174
アセンブラ …………37
アセンブリ言語 …………27
アセンブル …………37
アセンブルリスト …………51
アドレシングモード …………34
アドレス …………12
アドレス転送命令 …………46
アドレスバス …………9
アンパック型10進数 …………106
イミディエート …………35
インストラクションポインタ …………10
インデックス …………35
インデックスレジスタ …………10
エクストラセグメント …………13
エコーバック …………197
演算幅の拡張 …………91
オーバーフローフラグ …………87
オブジェクトプログラム …………27
オフセット …………12
オペランド …………27

◆か行◆

外部手続き …………154
加算 …………82
仮パラメータ …………189
間接アドレス …………53
完全なセグメント定義 …………33
簡略化セグメント疑似命令 …………48
簡略化セグメント定義 …………33
機械語 …………26, 51
疑似命令 …………28
キャリフラグ …………88
減算 …………84
後方参照 …………136
コードセグメント …………13

◆さ行◆

再配置可能 …………17
サインフラグ …………87
サブルーチン …………141
算術演算命令 …………30
実効アドレス …………58
実パラメータ …………189
シフトJISコード …………216
シフト命令 …………124
条件付分岐命令 …………134
乗算 …………96
除算 …………100
スタック …………68
スタックセグメント …………13
スタックトップ …………69
スタックポインタ …………10
スタック命令 …………46
ストリング …………16, 35, 162
ストリング命令 …………16, 30
制御分岐命令 …………30
セグメントオーバライドプリフィックス …36
セグメントレジスタ …………10
絶対番地 …………135
ゼロフラグ …………87
前方参照 …………136
相対番地 …………135
ソース …………28, 47
ソースアドレス …………17

ソースオペランド……83
ソースプログラム……27

◆た行◆

ダブルワード……15
単純化したアドレスモード……59
直接メモリ……35
ディスティネーション……28，47
ディスティネーションアドレス……17
ディスティネーションオペランド…83
ディレクションフラグ……88
ディレクティブ……28
ディレクトリ……40
データ移動命令……46
データ交換……65
データセグメント……13
データ転送命令……17，30
データバス……9
テーブル……74
手続き……141
デバッガ……272
デバッグ……272
転送先……47
転送元……47

◆な行◆

ニーモニック……27
入出力ポート……174
入出力命令……46

◆は行◆

バイト……15
パック型10進数……106
パリティフラグ……88
ビット……15
ビット操作命令……30
プッシュ……68
物理アドレス……14
フラグ……80
フラグ転送命令……46
フラグレジスタ……10
プリフィックス……17
ブレイクポイント……272，273
プロシージャ……141
プロセス……199
プロセッサ制御命令……30
ベースド……35
ベースドインデックス……35
ベースポインタ……10
補助キャリフラグ……87
ポップ……68

◆ま行◆

マイクロプロセッサ……8
マクロアセンブラ……40
無条件分岐命令……134
無符号……17
命令……9
命令キュー……9
命令セット……16
命令長……26
メモリ空間……9
メモリマップドI/O……175
モジュール……187
文字列……162

◆や行◆

有符号……17

◆ら行◆

ラベル……28，63，135
リンカ……37
リンク……37
ループ命令……134
レジスタ……10，35
レジスタ間接……35
ローテート命令……124

◆わ行◆

ワード……15
割り込み……176
割り込み処理……176
割り込み命令……178

命令および疑似命令

@data ……52
code ……48,52,184
data ……48,52,184
mode l ……52,183
stack ……99,184
? ……56-60
aaa ……106-111
aad ……106,107,116
aam ……106,107,114
aas ……106,107,112
adc ……81,92,93,107
add ……81-86,91-93
and ……120-122
assume ……33,184,185
call ……146,147
cbw ……102,103
clc ……89
cld ……89
cli ……177
cmc ……88,89
cmp ……89-91
cs ……140,184
cwd ……102-105
daa ……106,107
das ……106,107
db ……48,185
dd ……186
dec ……94-96
div ……101
ds ……184
dup ……60,188
dw ……48,186
endm ……188
endp ……186
equ ……29,188
idiv ……101-105
imul ……98
in ……174,175
inc ……94-96
include ……98
int ……178,196
into ……178
j? ……141-143
jmp ……134-139
label ……187
lea ……58,61,74,
lods ……162-167
macro ……187-189
mov ……46-64
movs ……162-167
mul ……97-99
nop ……138
not ……120,121
offset ……55,61,62,188
or ……120-122
out ……174-175
pop ……68-74,144-145
popf ……71
proc ……151,186
ptr ……94,136-138
public ……187
push ……68-74,144,145
pushf ……71
rcl ……124-129
rcr ……124-129
rep ……166
repne ……168
ret ……146,147
rol ……124-129
ror ……124-130
sal ……124-128
sar ……124-128
sbb ……81,93,107
shl ……124-128
shr ……124-128
ss ……184
stc ……88,89
std ……88,89
sti ……177
stos ……162-167
sub ……81-86,93
test ……120-122
xchg ……65-67
xlat ……74-76
xor ……120-123

■著者略歴
伊原充博（いはら　あつひろ）
1966年3月　東京都立工業高等専門学校電気工学科卒業
1969年3月　芝浦工業大学電気工学科卒業
1991年7月　東京都立工業高等専門学校電気工学科教授
1996年4月　同校電子情報工学科教授
著書に『よくわかるディジタルIC回路の基礎』（技術評論社・共著）がある．
小林弘幸（こばやし　ひろゆき）
1992年3月　東京都立大学工学部電気工学科卒
1997年3月　東京都立大学大学院工学研究科博士課程修
1997年4月　東京都立工業高等専門学校電気工学科講師
2000年4月　同校電気工学科助教授・工学博士

表紙デザイン◆小倉一夫
本文レイアウト◆株式会社アイテック（逸見育子）
企画◆金沢修
編集◆今村恵

◉注意◉
本書の内容に関するご質問は，すべて書面またはFAX（03-3225-5471）にて，下記住所第3編集部宛にお送りくださるようお願いいたします．
電話でのお問い合わせにはいっさいお答えできませんので，あらかじめご了承ください．

8086マクロアセンブラ入門（にゅうもん）

平成14年4月25日　初　版　第1刷発行

著　者　伊原（いはら）　充博（あつひろ）・小林（こばやし）　弘幸（ひろゆき）
発行者　片岡　巌
発行所　株式会社技術評論社
　　　　東京都新宿区愛住町8番地8
　　　　電話　03-3225-2300　販売促進部
　　　　　　　03-5366-8248　編集部
印刷／製本　日経印刷株式会社

定価は表紙に表示してあります．

ISBN4-7741-1442-1 C3055
Printed in Japan

ISBN4-7741-1442-1 C3055 ¥2180E

定価(本体2180円+税) M-code 403201